



無料電子ブック

学習

Java Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#java

.....	1
1: Java	2
.....	2
Java Edition	2
Java	2
Java	3
.....	3
.....	3
.....	3
.....	4
Examples.....	4
Java.....	4
Hello World	6
2: Apache Commons Lang	11
Examples.....	11
equals.....	11
hashCode.....	11
toString.....	12
3: BigDecimal	14
.....	14
Examples.....	14
BigDecimal.....	14
BigDecimals.....	14
BigDecimal.....	14
1.	14
2.Subtraction	15
3.	15
4.	15
5.RemainderModulus	16
6.Power	16

7.Max	16
8.Min	17
9.	17
10.	17
floatBigDecimal	17
BigDecimal.valueOf	18
BigDecimal110	19
4: BigInteger	20
	20
	20
	20
Examples	20
	20
BigInteger	22
BigInteger	23
BigInteger	25
BigInteger	26
5: BufferedWriter	28
	28
	28
Examples	28
	28
6: ByteBuffer	30
	30
	30
Examples	30
- ByteBuffer	30
-	30
- DirectByteBuffer	31
7: C ++	33
	33

.....	33
.....	33
C ++	33
Java.....	33
.....	33
C ++	33
Java.....	34
.....	34
C ++	34
Java.....	34
.....	34
.....	35
/.....	35
.....	35
.....	36
.....	36
C ++	36
.....	37
java.lang.Object.....	37
JavaC ++	37
Java.....	37
C ++	37
.....	37
Examples.....	38
.....	38
C ++	38
Java.....	38
.....	39
.....	39

C ++	39
Java	39
.....	39
C ++	39
Java	40
.....	40
C ++	40
Java	40
.....	40
C ++	40
Java	41
.....	41
.....	42
C ++	42
Java	42
.....	42
.....	42
C ++	42
Java	42
.....	42
C ++	43
Java	43
.....	43
C ++	43
Java	43
8: EnumSet	44
.....	44
Examples	44
Enum	44
9: Enum	45

.....	45
Examples.....	45
Enum.....	45
10: ExecutorExecutorServiceThreads.....	46
.....	46
.....	46
Examples.....	46
-	46
ThreadPoolExecutor.....	46
-	48
.....	49
.....	49
.....	49
.....	49
.....	50
submitexecute.....	51
.....	52
ExecutorService.....	54
ExecutorService.....	56
.....	57
11: FileUpload to AWS.....	59
.....	59
Examples.....	59
s3.....	59
12: FTP.....	62
.....	62
.....	62
Examples.....	62
FTP.....	62
13: HttpURLConnection.....	68
.....	68
Examples.....	68

URLString.....	68
POST.....	69
.....	69
.....	70
.....	70
.....	71
.....	71
.....	71
14: InputStreamsOutputStreams.....	72
.....	72
.....	72
Examples.....	72
InputStream.....	72
OutputStream.....	72
.....	73
.....	74
.....	74
.....	74
.....	74
DataInputStream.....	75
15: Java Edition.....	77
Examples.....	77
Java SE JREJava SE JDK.....	77
Java.....	77
Java.....	77
Oracle HotspotOpenJDK.....	78
Java EEJava SEJava MEJavaFX.....	78
Java.....	79
Java SE.....	79
Java EE.....	79
Java ME.....	79

Java FX	80
Java SE	80
Java SE	80
Java SE	81
16: Java SE 7	83
	83
	83
Examples	83
Java SE 7	83
	83
try-with-resources	84
	84
	84
switch	85
17: Java SE 8	86
	86
	86
Examples	86
Java SE 8	86
18: JavaStandard Edition	87
	87
Examples	87
WindowsPATHJAVA_HOME	87
	87
	87
	88
Java SE	88
Java	89
Java	90
LinuxJava JDK	90
	90
Oracle Java RPM	92

WindowsJava JDKJRE.....	92
MacOSJava JDK.....	93
LinuxJava.....	94
.....	94
.....	95
.....	95
.....	95
Linux.....	96
tarLinuxoracle java.....	97
.....	98
19: java.util.Objects.....	99
Examples.....	99
NULL.....	99
.....	99
not null.....	99
APIObjects.nonNull.....	99
20: JavaBean.....	100
.....	100
.....	100
.....	100
Examples.....	100
Java Bean.....	101
21: Java.....	102
Examples.....	102
.....	102
.....	103
.....	103
22: Java.....	105
.....	105
.....	105
.....	106

Examples.....	106
.....	106
.....	106
.....	107
.....	108
.....	108
Javadocs.....	109
.....	110
.....	111
23: Java	112
Examples.....	112
JSONPOJO.....	112
24: Java - 'java' 'javaw'	113
.....	113
.....	113
Examples.....	113
JAR.....	113
Java.....	113
HelloWorld.....	114
.....	114
.....	114
JavaFX.....	115
'java'.....	115
"".....	115
"".....	116
"<name>".....	117
.....	117
Java.....	117
.....	118
POSIX.....	119
Windows.....	119
Java.....	120

-D	120
.....	120
.....	120
VM	121
25: Java - 'javac'	123
.....	123
Examples	123
'javac' -	123
.....	123
.....	124
javac	125
javac	125
.....	126
Java	126
Java	126
.....	126
26: Java	128
.....	128
.....	128
Examples	128
TCP	128
27: Java2D	132
.....	132
Examples	132
1Java	132
2	134
28: Java	135
.....	135
.....	135
Examples	135
JAR	135

JARWAREAR.....	136
MavenJARWAR.....	137
AntJARWAREAR.....	137
IDEJARWAREAR.....	137
jarJARWAREAR.....	137
Java Web Start.....	137
.....	138
JNLP.....	138
Web.....	139
.....	139
Web Start.....	140
UberJAR.....	140
"jar"UberJAR.....	140
MavenUberJAR.....	141
UberJAR.....	141
29: Java	142
.....	142
.....	142
Examples.....	142
nashornjavascript.....	142
30: Java	145
Examples.....	145
JNA.....	145
JNA	145
.....	145
.....	146
31: Java	147
.....	147
.....	147
Examples.....	147
JavaC ++.....	147

Java.....	147
C ++.....	148
.....	149
C ++Java.....	149
Java.....	149
C ++.....	150
.....	150
.....	150
.....	151
.....	151
32: JavaJSON.....	153
.....	153
.....	153
Examples.....	153
JSON.....	153
JSON.....	154
optXXXgetXXX.....	154
JSONGson.....	155
JSON To ObjectGson.....	155
JSON.....	155
Jackson Object Mapper.....	156
.....	156
ObjectMapper.....	156
.....	156
.....	157
JSON.....	157
JSON Builder -	157
JSONObject.NULL.....	158
JsonArrayJavaGson.....	158
JacksonJSON.....	159
JSON.....	159
.....	159

.....	160
JSON	160
.....	160
.....	160
.....	160
.....	160
33: Java - NullPointerException	162
.....	162
Examples.....	162
- NullPointerException.....	162
Pitfall - null.....	163
-	164
"a" "b"null.....	165
null.....	165
.....	165
.....	165
.....	165
Pitfall - null.....	165
- I / OI / O.....	166
- NodaPointerException "Yoda".....	167
34: Java -	169
Examples.....	169
wait/ notify.....	169
.....	169
.....	169
/.....	170
- java.lang.Thread.....	170
-	171
-	172
.....	173
.....	

.....174

.....175

.....175

.....176

35: Java - 178

.....178

.....178

Examples.....178

- 178

.....178

- 179

- 'new'.....180

- String.....180

- System.gc.....180

- 182

- 183

- size.....183

- 183

PatternMatcher.....184

findmatch.....185

.....185

.....186

- ==.....187

.....187

'intern'.....187

.....187

.....188

.....188

- /.....188

.....189

.....190

.....	191
Java.....	191
36: Java -	192
.....	192
Examples.....	192
-	192
- RuntimeException.....	193
- ThrowableExceptionErrorRuntimeException.....	194
"throws"ThrowableException.....	195
- InterruptedException.....	195
-	197
-	198
- `Throwable`.....	199
37: Java -	200
.....	200
.....	200
Examples.....	200
-	200
-	200
-	201
-	202
-	204
- 8.....	205
-	206
- '=='.....	207
-	208
'assert'.....	208
.....	209
38: Java.....	210
Examples.....	210
.....	210
.....	210

Java.....	211
39: Java.....	213
.....	213
Examples.....	213
URLClassLoader.....	213
40: Java.....	217
.....	217
Examples.....	217
.....	217
.....	217
.....	218
.....	218
.....	219
.....	219
PrintJobEvent pje.....	221
.....	221
41: Java.....	223
.....	223
Examples.....	223
.....	223
.....	224
.....	224
.....	225
.....	225
.....	225
.....	226
.....	226
.....	226
.....	227
.....	227
2 'volatile'.....	228

3	229
.....	229
42: Java	231
.....	231
Examples	231
.....	231
1	231
GC	232
.....	232
C ++ -	232
Java -	233
.....	233
.....	234
PermGen	235
Java	236
.....	236
.....	237
43: JavaJVM	238
Examples	238
.....	238
44: Java	239
.....	239
Examples	239
.....	239
.....	241
.....	242
IEEE	242
45: JAXB	244
.....	244
.....	244
.....	244

.....	244
Examples.....	244
XML.....	244
XML.....	245
XmlAdapterxml.....	246
/XML@XmlAccessorType.....	247
/XML.....	249
XmlAdapter.....	249
.....	249
.....	249
.....	250
XML.....	251
.....	252
XMLJava.....	252
XmlAdapter.....	253
46: JAXP APIXML.....	254
.....	254
DOM.....	254
SAX.....	254
StAX.....	255
Examples.....	255
DOM API.....	255
StAX API.....	256
47: JAX-WS.....	259
Examples.....	259
.....	259
48: JMX.....	260
.....	260
Examples.....	260
MBean.....	260
49: JNDI.....	265
Examples.....	265

JNDIRMI.....	265
50: JShell.....	270
.....	270
.....	270
.....	270
.....	270
Examples.....	270
JShell.....	270
JShell.....	271
JShell.....	271
.....	271
.....	271
.....	271
.....	272
51: JVM.....	274
.....	274
Examples.....	274
1.0.....	274
JVMTI.....	276
Agent_OnLoad.....	277
52: JVM.....	278
.....	278
Examples.....	278
-XX.....	278
-XXallocClearChunks.....	278
-XXallocClearChunkSize.....	279
-XXcallProfiling.....	279
-XXdisableFatSpin.....	279
-XXdisableGCHururistics.....	279
-XXdumpSize.....	280
-XXexitOnOutOfMemory.....	280
53: log4j / log4j2.....	281

.....	281
.....	281
.....	281
Log4j 1	281
Examples.....	281
Log4j.....	281
Log4jJava.....	283
.....	283
log4j2.xml.....	284
log4j 1.x2.x.....	284
- DB.....	285
log4j 1.x.....	286
54: MultiThreadedThreadPoolExecutor	288
.....	288
Examples.....	288
.....	288
.....	290
Lambda.....	292
55: Nashorn JavaScript	295
.....	295
.....	295
.....	295
Examples.....	295
.....	295
.....	296
JavaScript.....	296
.....	296
.....	297
NashornJavaScriptJava.....	297
.....	298
.....	299
56: NIO -	300

.....	300
Examples.....	300
OP_CONNECT.....	300
57: NumberFormat.....	302
Examples.....	302
NumberFormat.....	302
58: RSA.....	303
Examples.....	303
OAEPGCM.....	303
59: ServiceLoader.....	308
.....	308
Examples.....	308
.....	308
.....	308
.....	308
META-INF / services / servicetest.Logger.....	309
.....	309
ServiceLoader.....	310
60: SortedMap.....	312
.....	312
Examples.....	312
.....	312
61: Stack-Walking API.....	314
.....	314
Examples.....	314
.....	314
.....	315
.....	315
62: static.....	317
.....	317
Examples.....	317

static.....	317
.....	317
.....	318
63: StringBuffer.....	320
.....	320
Examples.....	320
String Buffer.....	320
64: StringBuilder.....	322
.....	322
.....	322
.....	322
Examples.....	322
nString.....	322
StringBufferStringBuilderFormatterStringJoiner.....	323
65: sun.misc.Unsafe.....	325
.....	325
Examples.....	325
sun.misc.Unsafe.....	325
bootclasspathsun.misc.Unsafe.....	325
.....	326
.....	326
66: ThreadLocal.....	328
.....	328
Examples.....	328
ThreadLocal Java 8.....	328
ThreadLocal.....	328
1.....	330
67: TreeMapTreeSet.....	332
.....	332
Examples.....	332
JavaTreeMap.....	332
JavaTreeSet.....	333

JavaTreeMap / TreeSet.....	333
TreeMapTreeSet.....	335
68: XJC.....	337
.....	337
.....	337
.....	337
.....	337
Examples.....	337
XSDJava.....	337
XSDschema.xsd.....	337
xjc.....	338
.....	338
package-info.java.....	340
69: XML XPath.....	341
.....	341
Examples.....	341
XMLNodeList.....	341
1XMLXPath.....	341
XPathXML.....	342
70: XOM - XML.....	344
Examples.....	344
XML.....	344
XML.....	346
71:	350
.....	350
.....	350
.....	350
Examples.....	350
.....	350
72:	351
.....	351
.....	

Examples.....351

.....351

GUI.....352

.....353

.....353

.....353

.....354

.....354

73: 356

.....356

.....356

Examples.....356

forIterable.....356

.....356

Iterable.....357

.....358

74: 359

.....359

.....359

Examples.....359

.....359

.....360

.....361

Generics.....361

.....364

.....365

.....366

.....366

.....367

.....368

.....368

.....	369
.....	369
.....	369
75:	371
.....	371
Examples.....	371
.....	371
MIDI.....	371
.....	373
.....	373
76:	375
.....	375
.....	375
Examples.....	375
intInteger.....	375
if.....	376
NullPointerException.....	377
.....	377
Integerint.....	378
77:	380
.....	380
.....	380
Examples.....	380
toString.....	380
equals.....	381
.....	383
hashCode.....	384
Arrays.hashCode.....	385
.....	386
waitnotify.....	387
getClass.....	388
clone.....	389

finalize	390
.....	391
78:	394
.....	394
Examples	394
.....	394
Clonable	394
.....	395
.....	396
.....	397
79:	398
.....	398
Examples	398
.....	398
80:	402
.....	402
.....	402
Examples	402
Optional	402
.....	402
.....	403
.....	404
.....	404
.....	405
.....	405
.....	405
81:	407
.....	407
.....	407
Examples	407
.....	407
.....	407

.....	407
.....	407
.....	408
.....	408
.....	408
.....	408
.....	408
Java.....	409
.....	409
.....	409
.....	409
.....	410
.....	410
.....	410
.....	411
.....	411
.....	411
.....	412
.....	413
.....	413
.....	413
.....	413
.....	414
.....	414
.....	414
.....	415
.....	415
.....	416
.....	416
.....	416
82:	418
.....	418

.....	418
Examples.....	418
.....	418
.....	419
83:	421
.....	421
Examples.....	421
.....	421
.....	421
AM / PM.....	421
.....	422
84: - Java Reflection.....	423
.....	423
Examples.....	423
ObjectgetClass.....	423
85:	424
.....	424
.....	424
Examples.....	424
.....	424
.....	424
.....	425
.....	426
.....	429
staticstatic final.....	430
.....	430
86:	434
.....	434
.....	434
Examples.....	434
.....	434
JAR.....	435

.....	435
.....	436
.....	436
.....	437
.....	437
.....	438
87:	440
.....	440
Examples.....	440
.....	440
.....	440
.class.....	441
88:	443
.....	443
Examples.....	443
.....	443
settergetter.....	444
.....	444
89:	447
.....	447
.....	447
.....	447
Examples.....	447
GWT ToolBase.....	447
.....	448
.....	448
2.....	449
""1.....	449
90:	451
.....	451
.....	451
Examples.....	452

ArrayList.....	452
.....	453
.....	453
Java Collections.....	453
Google Guava Collections.....	453
.....	453
Java Collections.....	454
Apache Commons Collections.....	454
Google Guava Collections.....	454
.....	455
.....	455
.....	455
for"Banana".....	455
for.....	456
.....	456
Iteratorwhile.....	456
.....	457
.....	457
.....	457
.....	458
removelf.....	458
.....	458
.....	459
.....	459
.....	460
.....	460
.....	461
.....	461
IteratorList.....	462
Iteratorfor-eachIterable.....	463
.....	464
.....	

subListint fromIndexint toIndex.....	465
fromIndextoIndex.....	466
fromKeytoKey.....	466
91:	467
.....	467
Examples.....	467
Java.....	467
92:	468
.....	468
.....	468
.....	468
Examples.....	468
.....	468
.....	469
.....	469
93:	470
.....	470
Examples.....	470
.....	470
.....	470
.....	472
ConcurrentHashMap.....	472
94:	473
.....	473
.....	473
Examples.....	473
.....	473
.....	474
.....	475
95: I / O.....	477

Examples.....	477
.....	477
BufferedReader.....	477
Scanner.....	477
System.console.....	478
.....	478
.....	479
.....	480
96:	482
.....	482
.....	482
.....	482
Examples.....	482
.....	482
.....	483
.....	484
.....	485
.....	486
AB.....	487
.....	487
`T` T T '.....	489
.....	490
.....	490
.....	490
.....	491
1.....	491
.....	492
.....	492
.....	492
.....	492
Genericsinstanceof.....	494
.....	

.....	496
.....	497

97: JIT 499

.....	499
.....	499
Examples.....	499
.....	499

98: 502

.....	502
Examples.....	502
Java.....	502
Gson.....	504
Jackson 2.....	505
.....	506
serialVersionUID.....	508
.....	509
.....	509
JacksonJSON.....	510

99: 513

.....	513
Examples.....	513
.....	513
.....	513
.....	514
Bill Pugh.....	514
.....	515

100: 519

Examples.....	519
.....	519
.....	519
.....	520

.....	520
101:	522
.....	522
.....	522
.....	522
Examples	522
Scanner	522
Scanner	522
Scanner	523
.....	523
.....	524
int	526
.....	526
102:	527
.....	527
.....	527
Examples	527
.....	527
.....	528
.....	529
.....	529
.....	530
toList()toSet()	530
ListSet	530
.....	532
.....	532
.....	533
h21	534
.....	535
.....	535
.....	536
.....	536

.....	536
.....	537
.....	538
.....	538
IntStreamString.....	539
.....	539
.....	540
.....	540
.....	540
IntStream.....	541
flatMap.....	541
.....	542
.....	543
.....	544
.....	544
Map.Entry.....	545
.....	546
.....	546
.....	546
.....	546
.....	547
.....	547
.....	547
String.....	550
103:	552
Examples.....	552
.....	552
.....	552
/.....	553
.....	553
/.....	554
104:	556
.....	

.....	556
Examples.....	556
JCE.....	556
.....	556
Java.....	556
.....	556
.....	557
.....	557
105:	558
Examples.....	558
HashSet.....	558
.....	558
HashSet -	558
LinkedHashSet -	558
TreeSet - compareTo()Comparator.....	559
.....	559
.....	560
.....	561
Set.....	562
106:	563
.....	563
Examples.....	563
.....	563
107:	564
.....	564
Examples.....	564
.....	564
.....	564
.....	565
.....	565
instanceof.....	565

108:	567
.....	567
.....	567
Examples.....	567
Deque.....	567
Deque.....	567
.....	568
Deque.....	568
109:	569
.....	569
.....	569
.....	569
.....	569
.....	569
.....	570
Examples.....	570
.....	570
.....	571
.....	572
.....	572
.....	573
.....	574
110:	576
.....	576
Examples.....	576
.....	576
.....	576
.....	576
.....	576
.....	577
-	577
InputStreamTrustStoreKeyStore.....	579

- Web.....	579
UDP/.....	580
.....	581
SSL.....	583
.....	584
.....	585
111:	586
Examples.....	586
.....	586
.....	586
.....	586
/.....	586
.....	587
ASMjar.....	587
ClassNode.....	590
jar.....	590
Javassist Basic.....	591
112:	593
.....	593
.....	593
Examples.....	593
.....	593
.....	593
113:	595
.....	595
Examples.....	595
.....	595
114:	596
.....	596
Examples.....	596
/.....	596
.....	

597	
2.....	597
2.....	598
java.util.BitSet.....	600
.....	600
115: I / O.....	602
.....	602
Examples.....	602
[].....	602
.....	602
[].....	602
Stream vs Writer / Reader API.....	603
.....	604
.....	605
.....	605
java.io.FileJava 7NIOjava.nio.file.Path.....	606
.....	606
.....	606
.....	606
.....	606
OutputStream.....	607
.....	607
.....	608
FileInputStream / FileOutputStream.....	608
.....	610
.....	610
InputStreamOutputStream.....	611
.....	611
.....	612
BufferedInputStream.....	613
.....	613

PrintStream.....	614
.....	615
.....	615
/.....	615
ZIP.....	616
.....	616
.....	617
116: /.....	618
Examples.....	618
Java/.....	618
117:	620
.....	620
.....	620
.....	620
Examples.....	621
int.....	621
.....	621
.....	622
boolean.....	623
.....	623
float.....	624
.....	625
char.....	626
.....	626
.....	628
.....	629
.....	629
.....	629
118:	632
.....	632
Examples.....	632
.....	632

.....	633
.....	633
BufferedImage.....	635
BufferedImage.....	636
BufferedImage.....	637
BufferedImage.....	637
119:	639
.....	639
Examples.....	639
Java<1.5.....	639
ProcessBuilder.....	639
.....	640
ch.vorburger.exec.....	640
Runtime.execProcessProcessBuilder.....	641
.....	641
.....	642
.....	643
120:	644
.....	644
.....	644
.....	644
Examples.....	645
.....	645
caveat.....	645
XML.....	647
121:	650
.....	650
Examples.....	650
JMH.....	650
122:	653
.....	653
.....	653

Examples.....	653
.....	653
123:	655
.....	655
.....	655
.....	655
.....	655
Examples.....	655
.....	655
.....	657
.....	657
.....	658
124:	659
.....	659
.....	659
Examples.....	659
.....	659
.....	659
.....	660
Java.....	660
.....	660
.....	662
.....	663
.....	663
.....	663
.....	664
.....	664
.....	665
.....	665
.....	665
.....	665

.....	666
.....	666
.....	667
.....	667
`return`.....	668
Java.....	670
-	671
.....	672
.....	673
.....	673
125:	675
.....	675
.....	675
.....	675
Examples.....	676
.....	676
.....	677
.....	679
.....	680
AB.....	681
2.....	681
.....	682
ArrayList.....	682
List.....	683
.....	684
.....	684
-	685
.....	685
.....	685
.....	685
AttributeList.....	686

CopyOnWriteArrayList.....	686
LinkedList.....	686
.....	687
RoleUnresolvedList.....	687
.....	687
.....	687
126: SET.....	688
.....	688
Examples.....	688
.....	688
127:	689
.....	689
.....	689
Examples.....	690
.....	690
.....	691
JAR.....	691
.....	691
.....	692
.....	692
get.....	692
128:	694
.....	694
Examples.....	694
1682.....	694
.....	694
.....	695
Unicode.....	696
.....	696
10.....	696
.....	696
.....	

.....	697
.....	697
.....	698
.....	698
Null.....	698
.....	699
10.....	699
.....	700
16.....	700
.....	700
.....	700
.....	701
129: API.....	702
.....	702
.....	702
.....	702
Examples.....	702
.....	702
.....	704
.....	704
.....	705
.....	705
.....	706
.....	706
.....	707
.....	708
Reflection API.....	708
.....	709
.....	710
Evil Java.....	711
130: RMI.....	714

.....	714
Examples.....	714
- JVMJVM.....	714
"".....	716
.....	716
.....	716
.....	717
RMI.....	720
.....	720
.....	721
.....	723
131:	724
.....	724
Examples.....	724
Java LinkedHashMap.....	724
132:	726
.....	726
.....	726
Java.....	726
Examples.....	726
.....	726
Java	726
.....	727
.....	727
.....	728
.....	728
Java ResourceBundle	728
.....	728
133:	729
.....	729
Examples.....	729
.....	

134: java.util.logging	730
Examples	730
.....	730
.....	730
.....	731
135: Java	734
.....	734
Examples	734
==	734
.....	735
.....	736
==	737
.....	739
.....	740
.....	740
.....	741
2	742
.....	742
.....	743
String	744
136:	746
.....	746
Examples	746
.....	746
.....	746
.....	747
137:	752
.....	752
.....	752
Examples	752
.....	752

.....	752
.....	753
.....	754
138:	755
.....	755
.....	755
Examples.....	755
.....	755
/.....	756
ThreadLocal.....	757
CountDownLatch.....	758
.....	759
.....	761
.....	762
.....	764
/.....	764
java.lang.Thread.....	766
/.....	768
/.....	770
/.....	772
.....	773
.....	774
2 `int`.....	775
.....	776
.....	777
.....	778
139:	781
.....	781
Examples.....	781
.....	781
.....	781
.....	782

.....	782
.....	784
apache-common lang3.....	784
140:	785
.....	785
Examples.....	785
Apache HashBagGuava HashMultisetEclipse HashBag.....	785
1. ApacheSynchronizedSortedBag	786
EclipseGCTreeBag	786
3. GuavaLinkedHashMultiset	787
.....	787
GuavaApacheEclipse.....	788
Nore	792
-	792
-	792
141:	798
.....	798
.....	798
Examples.....	798
try-catch.....	798
1.....	798
.....	799
.....	800
.....	800
.....	801
.....	802
try-with-resources.....	803
.....	803
try-with-resource.....	804
try-with-resource.....	804
.....	805
try-with-resourcetry-catch-finally.....	805

.....	806
.....	807
.....	807
.....	809
.....	810
InterruptedException.....	810
Java -	812
.....	812
.....	813
.....	814
try catch.....	816
.....	817
.....	817
.....	818
.....	818
.....	819
try-finallytry-catch-finally.....	819
Try-finally.....	819
try-catch-finally.....	820
'throws'.....	821
.....	821
.....	821
142:	823
Examples.....	823
SecurityManager.....	823
ClassLoader.....	823
.....	824
DeniedPermission.....	825
DenyingPolicy.....	829
.....	831
143:	833
.....	833

.....	833
.....	833
.....	833
.....	833
Examples.....	834
.....	834
.....	835
.....	836
.....	838
.....	839
.....	839
.....	839
.....	840
144:	842
.....	842
.....	842
.....	842
.....	842
Java.....	842
Examples.....	843
.....	843
N.....	843
1N.....	844
N.....	844
.....	844
.....	845
.....	845
StackOverflowError.....	846
.....	846
.....	846
.....	846
Java.....	848

Java.....	849
145:	851
.....	851
.....	851
.....	851
.....	851
.....	851
Examples.....	851
.....	851
.....	855
.....	857
.....	858
.....	859
.....	859
enum.....	860
.....	861
.....	861
.....	861
.....	862
.....	863
.....	863
name(name).....	863
toString().....	864
.....	864
.....	864
Enum.....	864
.....	866
.....	866
Enum.....	868
146:	870
.....	870
.....	870

Examples.....	870
-	870
147:	873
.....	873
.....	873
.....	873
Examples.....	873
.....	873
.....	874
.....	875
.....	876
148:	877
Examples.....	877
.....	877
.....	877
149:	878
Examples.....	878
.....	878
150:	880
.....	880
.....	880
Examples.....	880
.....	880
.....	881
.....	881
.....	882
.....	882
.....	883
151:	884
.....	884
.....	884

Examples.....	884
.....	884
.....	885
Java 8.....	886
.....	888
.....	888
.....	889
Map <XY>Map <YZ>Map <XZ>.....	890
.....	891
NULL.....	891
.....	891
.....	894
HashMap.....	895
.....	896
.....	896
152:	899
.....	899
Examples.....	899
If / Else If / Else.....	899
For Loops.....	899
While.....	900
do ... while.....	901
For Each.....	901
If / Else.....	902
.....	903
.....	904
.....	905
.....	905
/.....	906
Java.....	906
153:	908
.....	908

Examples.....	908
varargs.....	908
Varargs.....	908
154:	910
.....	910
.....	910
Examples.....	910
.....	910
.....	912
.....	913
.....	914
.....	915
155:	919
.....	919
Examples.....	919
SealedObjectjavax.crypto.SealedObject.....	919
SignedObjectjava.security.SignedObject.....	919
156:	921
.....	921
Examples.....	921
.....	921
CompletableFuture.....	922
157:	923
Examples.....	923
.....	923
158:	924
.....	924
.....	924
Examples.....	924
.....	924
.....	925
.....	926

.....	926
.....	927
.....	927
.....	927
.....	928
.....	929
.....	929
159:	930
.....	930
Examples	930
.....	930
160:	932
Examples	932
PriorityQueue	932
FIFOLinkedList	932
.....	933
.....	933
Stack API	933
.....	933
BlockingQueue	934
.....	935
.....	936
.....	936
.....	937
161:	938
.....	938
Examples	938
.....	938
162:	939
Examples	939
UTF-8	939

UTF-8.....	939
UTF-8.....	940
163:	941
.....	941
.....	941
Examples.....	942
.....	942
==	943
switch	943
.....	943
.....	944
.....	944
/.....	945
.....	946
.....	947
.....	948
n.....	948
.....	949
toString.....	949
.....	950
.....	953
.....	953
.....	954
StringBuilders.....	955
.....	956
.....	956
.....	956
.....	957
.....	957
.....	957
.....	957
.....	957

.....	958
.....	960
164:	961
.....	961
Examples.....	961
StringTokenizer.....	961
StringTokenizer ".....	961
165:	962
Examples.....	962
String.....	962
/.....	962
Base64/.....	963
.....	964
`InputStream`` String`.....	965
.....	965
166:	968
.....	968
Examples.....	968
.....	968
.....	968
167: I / O	969
.....	969
Examples.....	969
.....	969
.....	969
.....	970
2	970
.....	970
.....	970
.....	970
.....	971

.....	971
MIME	971
.....	972
.....	972
168:	973
.....	973
.....	973
.....	973
Examples.....	974
Date.....	974
Date.....	974
.....	974
beforeaftercompareToequals	974
isBeforeisAftercompareToequals	976
Java 8	976
Java 8	977
.....	978
.....	978
.....	979
Date.....	979
.....	980
Java 8LocalDateLocalDateTime.....	980
java.util.Date.....	981
java.util.Datejava.sql.Date.....	982
.....	982
169: java.time*	984
Examples.....	984
.....	984
.....	984
.....	985
.....	985

Date Time API.....	985
.....	987
2LocalDate.....	988
170:	989
.....	989
Examples.....	989
Java Runtime Library.....	989
171:	992
.....	992
.....	992
.....	992
.....	992
.....	992
.....	992
.....	992
Examples.....	993
.....	993
regex.....	994
.....	994
.....	995
.....	995
.....	996
172:	997
.....	997
.....	997
Examples.....	997
Comparable.....	997
.....	1000
.....	1001
.....	1001
compareTo.....	1001
.....	

1002

.....	1003
.....	1003

173:	1005

.....	1005
.....	1005
.....	1005

.....	1005

Examples.....	1005
.....	1005
.....	1009
.....	1009
.....	1009

.....	1009

@.....	1010
.....	1010

@.....	1011
.....	1011

@Documented.....	1012

@.....	1012

@.....	1012
.....	1012
.....	1013
.....	1014
.....	1014
.....	1015

.....	1015

.....	1016

.....	1017

.....	1018

javac	1018

IDE	1019
.....	1019
.....	1019
.....	1019
this.....	1020
.....	1021
174:	1022
.....	1022
Examples.....	1022
-	1022
.....	1022
175:	1025
.....	1025
.....	1025
Examples.....	1025
+.....	1025
.....	1026
+ - */.....	1027
.....	1027
.....	1028
.....	1029
.....	1029
INFNAN.....	1029
===.....	1030
==!=.....	1030
==!=.....	1031
==!=.....	1031
NaN.....	1032
/++ / -	1032
:).	1033
.....	1033

.....	1033
^.....	1034
.....	1035
.....	1035
=+ = - =* =/ ==<< ==>> ==>>> == ^= =.....	1036
&& 	1038
- &&.....	1038
- &&.....	1039
<<>>>>>.....	1039
- >.....	1040
<<=>> =.....	1041
176:	1043
.....	1043
.....	1043
.....	1043
Examples.....	1043
.....	1043
.....	1043
2LocalTime.....	1044
.....	1045
177: AppDynamicsTIBCO BusinessWorks	1047
.....	1047
Examples.....	1047
AppdynamicsBW.....	1047
*** **	1047
178:	1048
.....	1048
.....	1048
.....	1048
Examples.....	1048
.....	1048
.....	1050

'final'	1051
.....	1051
.....	1051
.....	1052
.....	1052
.....	1053
.....	1054
.....	1055
.....	1056
.....	1057
"Is-a" "Has-a"	1059
.....	1062
179: JAR	1064
.....	1064
Examples	1064
Jar	1064
jarJar	1064
JarURL	1066
180:	1067
Examples	1067
.....	1067
PreferenceChangeEvent	1067
NodeChangeEvent	1067
.....	1068
.....	1069
.....	1069
.....	1070
.....	1071
.....	1072
.....	1072
.....	1072
181:	1074
.....	

1074	
Examples.....	1074
BufferedReader.....	1074
.....	1074
BufferedReader.....	1074
BufferedReader.....	1074
BufferedReader.readLine.....	1074
.....	1074
StringWriter.....	1075
182:	1076
.....	1076
.....	1076
Examples.....	1076
javap.....	1076
183:	1084
Examples.....	1084
.....	1084
184:	1085
.....	1085
.....	1085
.....	1085
Examples.....	1085
.....	1085
.....	1085
.....	1086
.....	1086
.....	1086
.....	1087
.....	1088
Java.....	1088
.....	1089
.....	

.....	1090
.....	1090
.....	1090
.....	1091
.....	1092
.....	1093
Arrays.asList.....	1094
.....	1094
Java.....	1095
ArrayIndexOutOfBoundsException.....	1096
.....	1096
.....	1097
.....	1098
.....	1098
.....	1100
for.....	1100
Object.clone.....	1101
Arrays.copyOf.....	1101
System.arraycopy.....	1101
Arrays.copyOfRange.....	1101
.....	1101
.....	1102
ArrayList.....	1102
System.arraycopy.....	1102
Apache Commons Lang.....	1102
.....	1103
.....	1103
.....	1104
.....	1104
Arrays.binarySearch.....	1104

Arrays.asList.....	1104
Stream.....	1104
.....	1105
org.apache.commons.....	1105
.....	1105
.....	1105
.....	1107
185:	1108
.....	1108
Examples.....	1108
.....	1108
.....	1109
.....	1109
.....	1111
.....	1111
.....	1112
strictfp.....	1112
.....	1114

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: [java-language](#)

It is an unofficial and free Java Language ebook created for educational purposes. All the content is extracted from [Stack Overflow Documentation](#), which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Java Language.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

1: Javaをいめる

Javaプログラミングは...

- いアプリケーションドメインでソフトウェアをするためにされており、のドメインにしたがけています。
- クラスベース そのオブジェクトはクラスでされます。クラスインスタンスは、にそのクラスでされたフィールドとメソッドをとっています「[クラスとオブジェクト](#)」を。これは、JavaScriptなどのクラスベースのとはです。
- き コンパイラは、がされることをコンパイルにチェックします。たとえば、メソッドが `String` のをとする、メソッドがびされると、そのはにはでなければなりません。
- オブジェクト Javaプログラムの、クラスインスタンス、つまりフィールドとるいデータをし、オブジェクトのインタフェースをにするメソッドです。
- **Portable** `javac` をしてのプラットフォームでコンパイルすることができ、のクラスファイルはJVMをつのプラットフォームでできます。

Javaは、アプリケーションに「いてみよう」WORAをにします。つまり、コンパイルされたJavaコードは、コンパイルをとせずにJavaをサポートするすべてのプラットフォームでできます。

JavaコードはJavaマシンJVMによってされるバイトコード `.class` ファイルにコンパイルされます。には、1つのJavaコンパイラでされたバイトコードは、なるのコンピュータであっても、すべてのJVMでじでするがあります。JVMは、のプログラムではにされるバイトコードのをネイティブマシンコマンドにコンパイルすることがあります。これは「ジャストインタイムJITコンパイル」とばれています。

Java Editionおよびバージョン

Sun / OracleによってされたJavaの「エディション」は3つあります。

- *Java Standard Edition* SEは、けにされたエディションです。
- *Java Enterprise Edition* EEには、Javaで「エンタープライズグレード」サービスをするためのさまざまながされています。Java EEは**となり**ます。
- *Java Micro Edition* MEは、Java SEのサブセットについており、られたリソースをつデバイスでのをとしています。

[Java SE / EE / MEエディション](#)にはのトピックがあります。

エディションにはのバージョンがあります。Java SEのバージョンはのとおりです。

Javaのインストール

[JavaStandard Editionのインストール](#)についてはのトピックがあります。

Javaプログラムのコンパイルと

のトピックがあります

- [Javaソースコードのコンパイル](#)
- [JARファイルのをむJavaデプロイメント](#)
- [Javaアプリケーションの](#)
- [クラスパス](#)

はですか

ここには、Javaプログラミングのとをするためのテーマへのリンクがあります。これらは、Javaプログラミングのであり、あなたをめてくれます。

- [Javaのプリミティブデータ](#)
- [Javaの](#)
- [Javaの](#)
- [Javaの](#)
- [Javaのクラスとオブジェクト](#)
- [Javaでの](#)
- [Javaコード](#)

テスト

Javaにはライブラリでのテストはサポートされていませんが、テストをサポートするようにされたサードパーティライブラリがあります。2つのもなユニットテストライブラリはのとおりです。

- [JUnit サイト](#)
- [TestNG サイト](#)

その

- Javaのためのデザインパターンがでわれている[デザインパターン](#)。
- AndroidプログラミングはAndroidでカバーされています。

- Java Enterprise Editionテクノロジーは、[Java EE](#)でカバーされています。
- OracleのJavaFXのはでわれている[JavaFX](#)の。

OracleはそのパブリックダウンロードサイトへのJava SEのなるをするバージョン **1.セクション** エンド・オブ・ライフです。なバグやセキュリティ、Java SEのなメンテナンスににアクセスするがあるおは、[Oracle Java SEのサポート](#)をしてなサポートをけることができます。

バージョン

Java SEバージョン	コード	エンド・オブ・ライフ ¹	
Java SE 9 アーリーアクセス	し		2017-07-27
Java SE 8	クモ		2014-03-18
Java SE 7	イルカ	2015-04-14	2011-07-28
Java SE 6	マスタング	2013-04-16	2006-12-23
Java SE 5		2009-11-04	2004104
Java SE 1.4	マーリン	2009114より	2002-02-06
Java SE 1.3	チョウゲンボウ	2009114より	2000-05-08
Java SE 1.2	び	2009114より	1998128
Java SE 1.1	し	2009114より	1997-02-19
Java SE 1.0	オーク	2009114より	1996121

Examples

のJavaプログラムをする

HelloWorld.javaというの[テキストエディタ](#)または[IDE](#)でしいファイルをしします。に、このコードブロックをファイルにりけてしします。

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Ideoneでライブ

Javaをpublic classとしてし [コンパイルエラー](#)をスローしないため、ファイルはが.javaのクラス

このでは `HelloWorld` とじでなければなりません。それに `public` アクセスもなければなりません。

では、Javaクラスはでまり、 **キャメルケース** ののがになりますであることがされています。このでは、アンダースコア `_` とドル `$` をすることをめします。

コンパイルするには、ターミナルウィンドウをき、 `HelloWorld.java` のディレクトリにします。

```
cd /path/to/containing/folder/
```

`cd` は `directory` をする `terminal` コマンドです。

のように、 `javac` いてファイルとをします。

```
$ javac HelloWorld.java
```

'javac' is not recognized as an internal or external command, operable program or batch file. というエラーをるのはかなり 'javac' is not recognized as an internal or external command, operable program or batch file. JDK をインストールしていても、IDE からプログラムをすることができます。eclipse パスは、デフォルトでにされていないので、など。

は、Windows でこれを、するには、まず `javac.exe` パスをしてみてください、それはおそらくあなたの `C:\Program Files\Java\jdk(version number)\bin` ます。それから、でそれをしてみてください。

```
$ C:\Program Files\Java\jdk(version number)\bin\javac HelloWorld.java
```

は `javac` をびしていたとき、のコマンドとじでした。その、あなたの OS は `javac` がどこにしているかをとっていました。は、パスをするはありません。これを `PATH` にするがあります

Windows XP / Vista / `PATH` をするには

- ・ コントロールパネル⇒システム⇒なシステム
- ・ 〇タブ⇒
- ・ 「システム」でにスクロールして「PATH」をします⇒

これをにすことはできませんのでしてください。まず、のパスをメモにコピーします。に `javac` への `PATH` をするには `javac` するフォルダをでし、アドレスバーをクリックしてコピーします。それは `c:\Program Files\Java\jdk1.8.0_xx\bin` ようにえるはずで

「」フィールドに、のすべてのディレクトリの **IN FRONT** とそれにくセミコロン;`;` をりけます。のエントリはしないでください。

```
Variable name : PATH
Variable value : c:\Program Files\Java\jdk1.8.0_xx\bin;[Existing Entries...]
```

これはするはずで

Linux ベースのシステムでは、 [ここでしてみてください](#)。

`javac` コマンドはJavaコンパイラをします。

コンパイラは `HelloWorld.class` というバイトコードファイルをし、JavaマシンJVMですることができます。Javaプログラミングコンパイラ `javac` は、Javaプログラミングでされたソースファイルを読み、`bytecode` クラスファイルにコンパイルします。オプションで、コンパイラは、`Pluggable Annotation Processing API` をして、ソースファイルおよびクラスファイルにあるアノテーションをすることもできます。コンパイラはコマンドラインツールですが、JavaコンパイラAPIをしてびすこともできます。

プログラムをするには、`java` に `main` メソッドこのでは `HelloWorld` をむクラスのをします。 `.class` がどのようにされているかにしてください

```
$ java HelloWorld
```

`java` コマンドはJavaアプリケーションをします。

これはコンソールにされます

こんにちは

のJavaプログラムをにコーディングしてビルドしました

Javaコマンド `java`、`javac` などがされるようにするには、のことはするがあります。

- JDKがインストールされている [Oracle](#)、[OpenJDK](#)などのソース
- あなたのがに [されている](#)

JVMがするコンパイラ `javac` とエグゼキュータ `java` をするがあります。インストールしたバージョンをするには、コマンドラインで `java -version` と `javac -version` とし `java -version`。プログラムのバージョンがにされます `1.8.0_73`。

Hello World プログラムをしくる

"Hello World"プログラムには、`HelloWorld` クラス、`main` メソッド、および `main` メソッドのステートメントでされるのファイルがまれています。

```
public class HelloWorld {
```

`class` キーワードは、`HelloWorld` クラスのクラスをします。すべてのJavaアプリケーションには、なくとも1つのクラス [クラスにする](#) がまれています。

```
    public static void main(String[] args) {
```

これはJVMがプログラムをできるエントリーポイントメソッドですと `public static void`

`main(String[])` シグネチャでされています。すべてのJavaプログラムには1つがです。それは

- `public` メソッドはプログラムのからでもどこからでもびすことができます。については、[Visibility](#)をしてください。
- `static` オブジェクトがし、クラスレベルでオブジェクトをせずにそれのできることをします。
- `void` をさないことをします。これはCやC++となり、`int`などのリターンコードがですJavaのは[System.exit\(\)](#)。

このなはをける

- へのとしてされるStringの `args` とばれる。えは、[コマンドライン](#)から

これのほとんどはJavaのエントリーポイントメソッドにです。

ではない

- `args` はなので、は `args` とばれますが、のでびすことができます。
- が `varargs` にされるので、そのパラメータが `String[] args` か [Varargs](#) `String... args` かどうかはありません。

1つのアプリケーションには、エントリーポイント `main` メソッドをむのクラスがするがあります。アプリケーションのエントリーポイントは、`java` コマンドのとしてされるクラスによってまします。

`main` メソッドのでは、のがされます。

```
System.out.println("Hello, World!");
```

このステートメントをごとにしましょう

System	これは、のが <code>java.lang</code> パッケージの <code>System</code> クラスをびすことをします。
.	これは「ドット」です。ドットはクラスメンバーへのアクセスをします。すなわちそのフィールドとそのメソッド。この、ドットをすると、することができます <code>out</code> のフィールドを <code>System</code> クラス。
out	これは、をむ <code>System</code> クラスの <code>PrintStream</code> のフィールドのです。
.	これはのドットです。このドットは、 <code>out</code> の <code>println</code> メソッドへのアクセスを <code>out</code> ます。
println	これは <code>PrintStream</code> クラスのメソッドのです。にこのメソッドは、パラメータのをコンソールにし、そのにをします。
(	このは、メソッドがアクセスされているフィールドではないことをし、 <code>println</code>

	メソッドにされるパラメータをします。
"Hello, World!"	これは、パラメータとして <code>println</code> メソッドにされる String リテラルです。のは、テキストをとしてります。
)	このは、 <code>println</code> メソッドにされるパラメータのクロージャをします。
;	このセミコロンはステートメントのわりをします。

Javaのステートメントは、セミコロン ; でわらなければなりません。

メソッドとクラスがじられます。

```

    } // end of main function scope
} // end of class HelloWorld scope

```

ここでは、オブジェクトのパラダイムをすのがあります。1つのはい、1つメンバーでフットボールチームをモデルしてみましょう。もっとくのことがあるかもしれませんが、たちがになったときにそれについてします。

まず、 `Team` クラスをしましょう

```

public class Team {
    Member member;
    public Team(Member member) { // who is in this Team?
        this.member = member; // one 'member' is in this Team!
    }
}

```

では、 `Member` クラスをしましょう

```

class Member {
    private String name;
    private String type;
    private int level; // note the data type here
    private int rank; // note the data type here as well

    public Member(String name, String type, int level, int rank) {
        this.name = name;
        this.type = type;
        this.level = level;
        this.rank = rank;
    }
}

```

なぜたちはここで `private` をしますかまあ、あなたのをりたいは、あなたのポケットにとってあなたのカードをきすわりに、あなたにねるべきです。この `private` は、のエンティティがあなたのにアクセスするのをぎます。 `private` メンバーは、ゲッターにすをしてのみすことができます。

すべてをまとめてゲッターとメインメソッドをしたは、のようになります。

```
public class Team {
    Member member;
    public Team(Member member) {
        this.member = member;
    }

    // here's our main method
    public static void main(String[] args) {
        Member myMember = new Member("Aurieel", "light", 10, 1);
        Team myTeam = new Team(myMember);
        System.out.println(myTeam.member.getName());
        System.out.println(myTeam.member.getType());
        System.out.println(myTeam.member.getLevel());
        System.out.println(myTeam.member.getRank());
    }
}

class Member {
    private String name;
    private String type;
    private int level;
    private int rank;

    public Member(String name, String type, int level, int rank) {
        this.name = name;
        this.type = type;
        this.level = level;
        this.rank = rank;
    }

    /* let's define our getter functions here */
    public String getName() { // what is your name?
        return this.name; // my name is ...
    }

    public String getType() { // what is your type?
        return this.type; // my type is ...
    }

    public int getLevel() { // what is your level?
        return this.level; // my level is ...
    }

    public int getRank() { // what is your rank?
        return this.rank; // my rank is
    }
}
```

```
Aurieel
light
10
1
```

イデオンでる

もう、`Test`クラスの`main`メソッドは、プログラムのエントリポイントです。`main`メソッドがなけ

れば、JavaマシンJVMにプログラムのをえることはできません。

1 - HelloWorldクラスはSystemクラスとのがないため、 publicデータにのみアクセスできます。

オンラインでJavaをいめるをむ <https://riptutorial.com/ja/java/topic/84/javaをいめる>

2: Apache Commons Lang

Examples

`equals` メソッドをする

オブジェクトの `equals` メソッドをにするに `equals`、`EqualsBuilder` クラスをします。

フィールドの

```
@Override
public boolean equals(Object obj) {

    if(!(obj instanceof MyClass)) {
        return false;
    }
    MyClass theOther = (MyClass) obj;

    EqualsBuilder builder = new EqualsBuilder();
    builder.append(field1, theOther.field1);
    builder.append(field2, theOther.field2);
    builder.append(field3, theOther.field3);

    return builder.isEquals();
}
```

をする

```
@Override
public boolean equals(Object obj) {
    return EqualsBuilder.reflectionEquals(this, obj, false);
}
```

`boolean` パラメータは、`equals` がフィールドをチェックするがあるかどうかをします。

をしていくつかのフィールドをける

```
@Override
public boolean equals(Object obj) {
    return EqualsBuilder.reflectionEquals(this, obj, "field1", "field2");
}
```

`hashCode` メソッドをする

オブジェクトの `hashCode` メソッドをにするには、`HashCodeBuilder` クラスをできます。

フィールドの

```
@Override
```

```
public int hashCode() {

    HashCodeBuilder builder = new HashCodeBuilder();
    builder.append(field1);
    builder.append(field2);
    builder.append(field3);

    return builder.hashCode();
}
```

をする

```
@Override
public int hashCode() {
    return HashCodeBuilder.reflectionHashCode(this, false);
}
```

booleanパラメータは、フィールドをするがあるかどうかをします。

をしていくつかのフィールドをける

```
@Override
public int hashCode() {
    return HashCodeBuilder.reflectionHashCode(this, "field1", "field2");
}
```

toString メソッドをする

オブジェクトのtoStringメソッドをにするには、ToStringBuilderクラスをできます。

フィールドの

```
@Override
public String toString() {

    ToStringBuilder builder = new ToStringBuilder(this);
    builder.append(field1);
    builder.append(field2);
    builder.append(field3);

    return builder.toString();
}
```

の

```
ar.com.jonat.lang.MyClass@dd7123[<null>,0,false]
```

にフィールドにをける

```
@Override
public String toString() {
```



```

    ToStringBuilder builder = new ToStringBuilder(this);
    builder.append("field1", field1);
    builder.append("field2", field2);
    builder.append("field3", field3);

    return builder.toString();
}

```

の

```
ar.com.jonat.lang.MyClass@dd7404[field1=<null>,field2=0,field3=false]
```

パラメータを使ってスタイルをすることができます

```

@Override
public String toString() {

    ToStringBuilder builder = new ToStringBuilder(this,
        ToStringStyle.MULTI_LINE_STYLE);
    builder.append("field1", field1);
    builder.append("field2", field2);
    builder.append("field3", field3);

    return builder.toString();
}

```

の

```
ar.com.bna.lang.MyClass@ebbf5c[
  field1=<null>
  field2=0
  field3=false
]
```

JSON、Classname、shortなどのスタイルがいくつかあります。

をして

```

@Override
public String toString() {
    return ToStringBuilder.reflectionToString(this);
}

```

あなたはスタイルをすることもできます

```

@Override
public String toString() {
    return ToStringBuilder.reflectionToString(this, ToStringStyle.JSON_STYLE);
}

```

オンラインでApache Commons Langをむ <https://riptutorial.com/ja/java/topic/3338/apache-commons-lang>

3: BigDecimal

き

`BigDecimal`クラスは、、、、スケール、め、、、ハッシュ、およびフォーマットのためのをします。 `BigDecimal`は、ののき10をします。このクラスは、をとするにするものとする。

Examples

`BigDecimal`オブジェクトはです

`BigDecimal`でしたい、`BigDecimal`オブジェクトはであるため、されたをするがあります

```
BigDecimal a = new BigDecimal("42.23");
BigDecimal b = new BigDecimal("10.001");

a.add(b); // a will still be 42.23

BigDecimal c = a.add(b); // c will be 52.231
```

`BigDecimal`sの

`BigDecimal`sをするには、 `compareTo`メソッドをするがあります。

```
BigDecimal a = new BigDecimal(5);
a.compareTo(new BigDecimal(0)); // a is greater, returns 1
a.compareTo(new BigDecimal(5)); // a is equal, returns 0
a.compareTo(new BigDecimal(10)); // a is less, returns -1
```

は、 `equals`メソッドをしないでください。なぜなら、2つの`BigDecimal`sは、がしく、 りもしいにのみしいとみなすから`equals`。

```
BigDecimal a = new BigDecimal(5);
a.equals(new BigDecimal(5)); // value and scale are equal, returns true
a.equals(new BigDecimal(5.00)); // value is equal but scale is not, returns false
```

`BigDecimal`による

このでは、`BigDecimal`sをしてなをするをします。

1.

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");
```

```
//Equivalent to result = a + b
BigDecimal result = a.add(b);
System.out.println(result);
```

12

2.Subtraction

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

//Equivalent to result = a - b
BigDecimal result = a.subtract(b);
System.out.println(result);
```

-2

3.

2つの`BigDecimal`を合わせると、はオペランドのスケールのとしいスケールになります。

```
BigDecimal a = new BigDecimal("5.11");
BigDecimal b = new BigDecimal("7.221");

//Equivalent to result = a * b
BigDecimal result = a.multiply(b);
System.out.println(result);
```

36.89931

のスケールがオーバーロードし`MathContext`のオペレータのためのルールをしたオブジェクト、の
とめモードを-。なめモードのについては、Oracleのマニュアルをしてください。

```
BigDecimal a = new BigDecimal("5.11");
BigDecimal b = new BigDecimal("7.221");

MathContext returnRules = new MathContext(4, RoundingMode.HALF_DOWN);

//Equivalent to result = a * b
BigDecimal result = a.multiply(b, returnRules);
System.out.println(result);
```

36.90

4.

は、のよりもしです。たとえば、のをえてみましょう。

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

BigDecimal result = a.divide(b);
System.out.println(result);
```

これで0.7142857142857143とたものがられるとしていますが、

java.lang.ArithmeticExceptionしないの。なのはありません。

これは、5で2をしたい、がになるとにうまくしますが、するとのをすにして `ArithmeticException` ます。のシナリオでは、にするをできないため、`BigDecimal`のスケールとめモードをするがあります。とめモードのについては、[Oracleのマニュアル](#)をしてください。

たとえば、はできる

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

//Equivalent to result = a / b (Upto 10 Decimal places and Round HALF_UP)
BigDecimal result = a.divide(b,10,RoundingMode.HALF_UP);
System.out.println(result);
```

0.7142857143

5.RemainderまたはModulus

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

//Equivalent to result = a % b
BigDecimal result = a.remainder(b);
System.out.println(result);
```

5

6.Power

```
BigDecimal a = new BigDecimal("5");

//Equivalent to result = a^10
BigDecimal result = a.pow(10);
System.out.println(result);
```

9765625

7.Max

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

//Equivalent to result = MAX(a,b)
BigDecimal result = a.max(b);
System.out.println(result);
```

7

8.Min

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

//Equivalent to result = MIN(a,b)
BigDecimal result = a.min(b);
System.out.println(result);
```

5

9.に

```
BigDecimal a = new BigDecimal("5234.49843776");

//Moves the decimal point to 2 places left of current position
BigDecimal result = a.movePointLeft(2);
System.out.println(result);
```

52.3449843776

10.にポイントをする

```
BigDecimal a = new BigDecimal("5234.49843776");

//Moves the decimal point to 3 places right of current position
BigDecimal result = a.movePointRight(3);
System.out.println(result);
```

5234498.43776

のにはさらにくのオプションとパラメータのみわけがありますたとえば、メソッドの6つのバリエーションがあります。このセットはなリストであり、いくつかのながまれています。

floatのわりに**BigDecimal**をう

がコンピュータメモリでされるのため、このをしたはになるがあります。のはとしてされます。

これのいはず。がなは、のタイプをするがあります。 Java 7はBigDecimalをします。

```
import java.math.BigDecimal;

public class FloatTest {

    public static void main(String[] args) {
        float accountBalance = 10000.00f;
        System.out.println("Operations using float:");
        System.out.println("1000 operations for 1.99");
        for(int i = 0; i<1000; i++){
            accountBalance -= 1.99f;
        }
        System.out.println(String.format("Account balance after float operations: %f",
accountBalance));

        BigDecimal accountBalanceTwo = new BigDecimal("10000.00");
        System.out.println("Operations using BigDecimal:");
        System.out.println("1000 operations for 1.99");
        BigDecimal operation = new BigDecimal("1.99");
        for(int i = 0; i<1000; i++){
            accountBalanceTwo = accountBalanceTwo.subtract(operation);
        }
        System.out.println(String.format("Account balance after BigDecimal operations: %f",
accountBalanceTwo));
    }
}
```

このプログラムのはのとおりです。

```
Operations using float:
1000 operations for 1.99
Account balance after float operations: 8009,765625
Operations using BigDecimal:
1000 operations for 1.99
Account balance after BigDecimal operations: 8010,000000
```

のが10000.00の、1.99の1000の、は8010.00になるとしてしています。 floatをすると、8009.77のがられますが、これはのにはできないほどです。 BigDecimalをすると、ながられます。

BigDecimal.valueOf

BigDecimalクラスには、にされる010などのキャッシュがまれます。 BigDecimal.valueOfメソッドは、したパラメータをつコンストラクタにしてされます。

```
BigDecimal a = BigDecimal.valueOf(10L); //Returns cached Object reference
BigDecimal b = new BigDecimal(10L); //Does not return cached Object reference

BigDecimal a = BigDecimal.valueOf(20L); //Does not return cached Object reference
BigDecimal b = new BigDecimal(20L); //Does not return cached Object reference

BigDecimal a = BigDecimal.valueOf(15.15); //Preferred way to convert a double (or float) into
a BigDecimal, as the value returned is equal to that resulting from constructing a BigDecimal
from the result of using Double.toString(double)
BigDecimal b = new BigDecimal(15.15); //Return unpredictable result
```

BigDecimalのゼロ、1または10の

BigDecimalは、ゼロ、1、および10ののプロパティをします。のをするわりに、これらをするこ
ことをおめします。

- `BigDecimal.ZERO`
- `BigDecimal.ONE`
- `BigDecimal.TEN`

なプロパティをすることで、なインスタンスをけることができます。また、「マジックナンバー」
」のわりにコードにリテラルがあります。

```
//Bad example:
BigDecimal bad0 = new BigDecimal(0);
BigDecimal bad1 = new BigDecimal(1);
BigDecimal bad10 = new BigDecimal(10);

//Good Example:
BigDecimal good0 = BigDecimal.ZERO;
BigDecimal good1 = BigDecimal.ONE;
BigDecimal good10 = BigDecimal.TEN;
```

オンラインでBigDecimalをむ <https://riptutorial.com/ja/java/topic/1667/bigdecimal>

4: BigInteger

き

`BigInteger` クラスは、プリミティブデータにはきすぎるきなをむにされます。たとえば、100は158です。これは、`long` よりもはるかにきくなります。 `BigInteger` は、Javaのプリミティブのすべて、および `java.lang.Math` からのするすべてのメソッド、およびのほとんどのにしたものをして `java.lang.Math`。

- `BigInteger variable_name = しい BigInteger "12345678901234567890";` //としての10
- `BigInteger variable_name = new BigInteger`
"`1010101101010100101010011000110011101011000111110000101011010010`"、2//としての2
- `BigInteger variable_name = new BigInteger "ab54a98ceb1f0800";`、16//としての16
- `BigInteger variable_name = しい BigInteger64、しい ランダム;` // 64ビットをしてをする
- `BigInteger variable_name = new BigInteger しい バイト [] {0、-85、84、-87、-116、-21、31、10、-46};` //のき2のビッグエンディアン
- `BigInteger variable_name = new BigInteger1、しい バイト [] {-85,84、-87、-116、-21、31、10、-46};` //ののなし2のビッグエンディアン

`BigInteger` はです。したがって、そのをすることはできません。たとえば、のために `sum` がされないため、はしません。

```
BigInteger sum = BigInteger.ZERO;
for(int i = 1; i < 5000; i++) {
    sum.add(BigInteger.valueOf(i));
}
```

を `sum` にして、させます。

```
sum = sum.add(BigInteger.valueOf(i));
```

Java SE 8

`BigInteger` のによると、 `BigInteger` では、 $-2^{2147483647}$ と $2^{2147483647}$ のすべてののがサポートされるはずでです。つまり、 `BigInteger` のビットは20をえることができます

Examples

`java.math.BigInteger` クラスは、Javaのすべてのプリミティブと、 `java.lang.Math` するすべてのメソッドにしてのをし `java.lang.Math`。 `java.math` パッケージはにできないため、なクラスをするに `java.math.BigInteger` をインポートするがあります。

`long`

または `int` を `BigInteger` するには、のようになります。

```
long longValue = Long.MAX_VALUE;
BigInteger valueFromLong = BigInteger.valueOf(longValue);
```

または、の

```
int intValue = Integer.MIN_VALUE; // negative
BigInteger valueFromInt = BigInteger.valueOf(intValue);
```

これらでであろう `intValue` のはとどまるように、のビットをして、のです。

の `String` を `BigInteger` するには

```
String decimalString = "-1";
BigInteger valueFromDecimalString = new BigInteger(decimalString);
```

のコンストラクタは、のにするためにされ `BigInteger` にされたでの `BigInteger`。

```
String binaryString = "10";
int binaryRadix = 2;
BigInteger valueFromBinaryString = new BigInteger(binaryString , binaryRadix);
```

Javaでは、`BigInteger` インスタンスへのバイトのもサポートされています。は、きおよびなしのビッグエンディアンエンコーディングのみができます。

```
byte[] bytes = new byte[] { (byte) 0x80 };
BigInteger valueFromBytes = new BigInteger(bytes);
```

これにより、のビットがビットとしてされるため、-128の `BigInteger` インスタンスがされます。

```
byte[] unsignedBytes = new byte[] { (byte) 0x80 };
int sign = 1; // positive
BigInteger valueFromUnsignedBytes = new BigInteger(sign, unsignedBytes);
```

これは、バイトがなしとしてされ、がに1ののにされるため、が128の `BigInteger` インスタンスをします。

のにはあらかじめされたがあります。

- `BigInteger.ZERO` - が「0」です。
- `BigInteger.ONE` - が「1」です。
- `BigInteger.TEN` - が「10」です。

`BigInteger.TWO` "2"もありますが、`private` ためコードですることはできません。

BigIntegersの

BigIntegersは、JavaのStringやのオブジェクトとするのと同じものをできます。

例えば

```
BigInteger one = BigInteger.valueOf(1);
BigInteger two = BigInteger.valueOf(2);

if(one.equals(two)){
    System.out.println("Equal");
}
else{
    System.out.println("Not Equal");
}
```

```
Not Equal
```

に、BigIntegersをするために==をしないでください

- ==をします。つまり、2つのがじオブジェクトをしているかどうか
- equals()メソッド2つのBigIntegerのをします。

たとえば、BigIntegersはのようになしないでください。

```
if (firstBigInteger == secondBigInteger) {
    // Only checks for reference equality, not content equality!
}
```

==がのみをチェックするので、しないにつながるがあります。のBigIntegerにじコンテンツがまわっていて、じオブジェクトをしていない、これはします。わりに、でしたように、equalsメソッドをしてBigIntegersをします。

BigIntegerと0,1,10のようなをすることもできます。

例えば

```
BigInteger reallyBig = BigInteger.valueOf(1);
if(BigInteger.ONE.equals(reallyBig)){
    //code when they are equal.
}
```

のように、compareTo()メソッドをして2つのBigIntegerをすることもできます。compareTo()は3つのをします。

- 0がしい。
- 1のが2のよりもきいの。
- -1のものが2のものよりもさいとき。

```

BigInteger reallyBig = BigInteger.valueOf(10);
BigInteger reallyBig1 = BigInteger.valueOf(100);

if(reallyBig.compareTo(reallyBig1) == 0){
    //code when both are equal.
}
else if(reallyBig.compareTo(reallyBig1) == 1){
    //code when reallyBig is greater than reallyBig1.
}
else if(reallyBig.compareTo(reallyBig1) == -1){
    //code when reallyBig is less than reallyBig1.
}

```

BigIntegerのの

BigIntegerはオブジェクトであるため、ののをしいBigIntegerインスタンスにりてるがあります。

$10 + 10 = 20$

```

BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("10");

BigInteger sum = value1.add(value2);
System.out.println(sum);

```

20

$10 - 9 = 1$

```

BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("9");

BigInteger sub = value1.subtract(value2);
System.out.println(sub);

```

1

$10/5 = 2$

```

BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("5");

BigInteger div = value1.divide(value2);
System.out.println(div);

```

2

$17/4 = 4$

```

BigInteger value1 = new BigInteger("17");
BigInteger value2 = new BigInteger("4");

BigInteger div = value1.divide(value2);

```

```
System.out.println(div);
```

4

$10 * 5 = 50$

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("5");

BigInteger mul = value1.multiply(value2);
System.out.println(mul);
```

50

$10^3 = 1000$

```
BigInteger value1 = new BigInteger("10");
BigInteger power = value1.pow(3);
System.out.println(power);
```

1000

り $106 = 4$

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("6");

BigInteger power = value1.remainder(value2);
System.out.println(power);
```

4

GCD 12 と 18 **GCD** は 6 です。

```
BigInteger value1 = new BigInteger("12");
BigInteger value2 = new BigInteger("18");

System.out.println(value1.gcd(value2));
```

6

2つのBigInteger

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("11");

System.out.println(value1.max(value2));
```

11

2つのBigInteger

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("11");

System.out.println(value1.min(value2));
```

10

BigIntegerのバイナリロジック

BigIntegerは`Number`でもできるバイナリロジックをサポートしています。すべてののに、メソッドをびすことでされます。

バイナリまたは

```
BigInteger val1 = new BigInteger("10");
BigInteger val2 = new BigInteger("9");

val1.or(val2);
```

11これは $10 \mid 9$

バイナリと

```
BigInteger val1 = new BigInteger("10");
BigInteger val2 = new BigInteger("9");

val1.and(val2);
```

8 $10 \& 9$

バイナリXOR

```
BigInteger val1 = new BigInteger("10");
BigInteger val2 = new BigInteger("9");

val1.xor(val2);
```

3 $10 \wedge 9$

RightShift

```
BigInteger val1 = new BigInteger("10");

val1.shiftRight(1); // the argument be an Integer
```

5 $10 \gg 1$

```
BigInteger val1 = new BigInteger("10");

val1.shiftLeft(1); // here parameter should be Integer
```

20 10 << 1

バイナリNot

```
BigInteger val1 = new BigInteger("10");  
  
val1.not();
```

5

NANDAnd-Not *

```
BigInteger val1 = new BigInteger("10");  
BigInteger val2 = new BigInteger("9");  
  
val1.andNot(val2);
```

7

ランダムBigIntegersの

BigIntegerクラスには、java.util.RandomインスタンスとBigIntegerつビットをするintをえられたランダムBigIntegersをするのコンストラクタがあります。そのいはいにです。のように
`BigInteger(int, Random)` コンストラクタをびすと、

```
BigInteger randomBigInt = new BigInteger(bitCount, sourceOfRandomness);
```

が0をむから2 bitCount これをまないまでのBigIntegerます。

これは、new BigInteger(2147483647, sourceOfRandomness)がnew BigInteger(2147483647, sourceOfRandomness)ながえられたすべてののBigIntegerすこともあることをします。

sourceOfRandomnessはあなたです。たとえば、new Random()はほとんどのです

```
new BigInteger(32, new Random());
```

ののをあきらめたいは、わりにnew SecureRandom ()うことができます

```
import java.security.SecureRandom;  
  
// somewhere in the code...  
new BigInteger(32, new SecureRandom());
```

クラスでオンザフライでアルゴリズムをすることもできます ロールアウトのRNGアルゴリズムは
そういつもあなたがたをき、まともであることがされたアルゴリズムをしてください、のランダム
であなたをわることにしてくださいBigIntegerであることを。

```
new BigInteger(32, new Random() {  
    int seed = 0;  
  
    @Override  
    protected int next(int bits) {  
        seed = ((22695477 * seed) + 1) & 2147483647; // Values shamelessly stolen from  
Wikipedia  
        return seed;  
    }  
});
```

オンラインでBigIntegerをむ <https://riptutorial.com/ja/java/topic/1514/biginteger>

5: BufferedWriter

- `new BufferedWriter()`; //デフォルトコンストラクタ
 - `BufferedWriter.write(int c)`; //のをきみます。
 - `BufferedWriter.writeString(String str)`; //をきみます。
 - `BufferedWriter.newLine()`; //りをきみます。
 - `BufferedWriter.close()`; // `BufferedWriter`をじます。
-
- あなたからのきみをしようとするとき `BufferedWriter` して `BufferedWriter.write()` した `BufferedWriter` して `BufferedWriter.close()` それをスローされます `IOException`。
 - `BufferedWriter(Writer)` コンストラクタは `IOException` スローしません。ただし、`FileWriter(File)` コンストラクタは `IOException` をする `FileNotFoundException` スローします。だから、`IOException` をキャッチすると `FileNotFoundException` もキャッチされますが、`FileNotFoundException` とかうことをするがないり、2のcatchはありません。

Examples

ファイルにテキストをく

このコードはをファイルにきみます。ライターをじることはです。だから、これは `finally` ブロックでわれます。

```
public void writeLineToFile(String str) throws IOException {
    File file = new File("file.txt");
    BufferedWriter bw = null;
    try {
        bw = new BufferedWriter(new FileWriter(file));
        bw.write(str);
    } finally {
        if (bw != null) {
            bw.close();
        }
    }
}
```

また、`write(String s)` は、がきまれたにをしないことにもしてください。それをれるには、`newLine()` メソッドをします。

Java SE 7

Java 7は、[java.nio.file](#) パッケージと [try-with-resources](#) をします。

```
public void writeLineToFile(String str) throws IOException {
    Path path = Paths.get("file.txt");
    try (BufferedWriter bw = Files.newBufferedWriter(path)) {
        bw.write(str);
    }
}
```


オンラインでBufferedWriterをむ <https://riptutorial.com/ja/java/topic/3063/bufferedwriter>

6: ByteBuffer

き

ByteBuffer クラスはJava 1.4でされ、バイナリデータのをにしました。これはプリミティブのデータですのににしています。これは、をにするだけでなく、よりいレベルでのbyte[] sのものにする

- byte [] arr = しいバイト[1000];
- ByteBuffer バッファ = ByteBuffer.wrap(arr);
- ByteBuffer バッファ = ByteBuffer.allocate(1024);
- ByteBuffer バッファ = ByteBuffer.allocateDirect(1024);
- バイト b = buffer.get;
- バイト b = buffer.get(10);
- ショート s = buffer.getShort(10);
- buffer.put(120);
- buffer.putChar('a');

Examples

ない - ByteBuffer の

ByteBuffer をするには、2つのがあります。1つは、もう1つは、もうすることができます。

すでにするbyte[] がある、ByteBuffer に「ラップ」してをできます。

```
byte[] reqBuffer = new byte[BUFFER_SIZE];
int readBytes = socketInputStream.read(reqBuffer);
final ByteBuffer reqBufferWrapper = ByteBuffer.wrap(reqBuffer);
```

これは、レベルのネットワークインタラクションをするコードのがあります

すでにしているbyte[] がないは、のようにバッファにりてられたにByteBuffer をすることができます

```
final ByteBuffer respBuffer = ByteBuffer.allocate(RESPONSE_BUFFER_SIZE);
putResponseData(respBuffer);
socketOutputStream.write(respBuffer.array());
```

コードパスがにで、システムメモリへのアクセスがな、ByteBuffer は#allocateDirect() をしてバッファをりてることさえできます。

な - バッファにデータをきむ

ByteBufferインスタンスがえられると、 々な `put` をして、プリミティブのデータをきむことができます。きないは、メソッドをしてデータをすると、データがされるインデックスがされますが、メソッドではにデータを`put`するインデックスがです。

どちらのも「」コールをします。なきさのバッファがあれば、それにじてのことができる。

```
buffer.putInt(0xCAFEBADE).putChar('c').putFloat(0.25).putLong(0xDEADBEEFCAFEBADE);
```

これはとです

```
buffer.putInt(0xCAFEBADE);
buffer.putChar('c');
buffer.putFloat(0.25);
buffer.putLong(0xDEADBEEFCAFEBADE);
```

`byte s`でするメソッドはなではないことにしてください。さらに、それはすこともだということにしByteBufferと`byte[]`して`put`。そののすべてのプリミティブはな`put`メソッドをとっています。

の`put*`をするときにされるインデックスは、に`byte`カウントされ`byte`。

ない - DirectByteBufferの

DirectByteBufferは、 `byte[]`がしないByteBufferなです。

このようなByteBufferをびすには、

```
ByteBuffer directBuffer = ByteBuffer.allocateDirect(16);
```

このでは、16バイトのメモリがりてられます。ダイレクトバッファのは、のガベージコレクトされたヒープのにすることがあります。

ByteBufferがかどうかは、のびしによってできます。

```
directBuffer.isDirect(); // true
```

DirectByteBufferのなは、のバッファリングをわずにJVMがりてられたメモリをネイティブにしようとするため、そのにされたをつByteBufferでされるがいことです。

リアルタイムのように、にするIOのDirectByteBufferをすることをおめします。

`array()`メソッドをしようとする、 `UnsupportedOperationException`がすることにするがあります。だからたちがそれにアクセスしようとするにByteBufferがバイトをとっているかどうかをするのはいです

```
byte[] arrayOfBytes;
if(buffer.hasArray()) {
    arrayOfBytes = buffer.array();
}
```

バイトバッファののは、JNIをするinteropです。ダイレクトバイトバッファは`byte[]`ではなくのメモリブロックをするため、ネイティブコードのポインタをしてそのメモリにアクセスすることができます。これにより、Javaとネイティブのデータのでマーシャリングするのトラブルやオーバーヘッドがされます。

JNIインタフェースは、バイトバッファをするいくつかのをしています。 [NIOサポート](#)。

オンラインでByteBufferをむ <https://riptutorial.com/ja/java/topic/702/bytebuffer>

7: C ++の

き

JavaとC ++はのです。このトピックは、JavaおよびC ++エンジニアのクイックリファレンスガイドとしてします。

のコンストラクトでされたクラス

のクラスでされる

C ++

ネストされたクラス[\[ref\]](#)むクラスへのがです

```
class Outer {
    class Inner {
    public:
        Inner(Outer* o) :outer(o) {}

    private:
        Outer*  outer;
    };
};
```

Java

[でない]ネストされたクラスクラスまたはメンバークラス

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
}
```

のクラスでにされる

C ++

ネストされたクラス

```
class Outer {
    class Inner {
        ...
    };
};
```

Java

ネストされたクラスメンバークラス [\[ref\]](#)

```
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
}
```

メソッドでされる

例えば、イベント

C ++

ローカルクラス [\[ref\]](#)

```
void fun() {
    class Test {
        /* members of Test class */
    };
}
```

[ラムダ](#)も

Java

ローカルクラス [\[ref\]](#)

```
class Test {
    void f() {
        new Thread(new Runnable() {
            public void run() {
                doSomethingBackgroundish();
            }
        }).start();
    }
}
```

オーバーライドオーバーロード

のOverriding vs Overloadingポイントは、C++とJavaのにされます。

- オーバーライドされたメソッドは、メソッドとじとをちます。
- オーバーロードされたメソッドはじですが、かなり、にしません。
- じとをち、りのがなる2つのメソッドはです。するStackoverflowのをしてください"なるり
のでJavaでオーバーロードする"- [1](#)。 [2](#)

とは、によってするなるクラスのオブジェクトがじメソッドびしになるをするです。ここにが
あります

- クラスメソッドとしてをつ
- SquareクラスとCircleクラスの2つのクラスは、エリアメソッドをしています
- SquareとSquareがびされます

C++では、メソッドによってがになります。Javaでは、メソッドはデフォルトではです。

Iの

オブジェクトのクリーンアップ

C++では、デストラクタをとしてして、ベースクラスのポインタがされたにサブクラスのデスト
ラクタがにびされるようにすることをおめします。

Javaでは、finalizeメソッドはC++のデストラクタとています。ただし、ファイナライザはできま
せんGCにしています。ベストプラクティス - にクリーンアップするには、「クローズ」メソッド
をします。

```
protected void close() {
    try {
        // do subclass cleanup
    }
    finally {
        isClosed = true;
        super.close();
    }
}

protected void finalize() {
    try {
        if(!isClosed) close();
    }
    finally {
        super.finalize();
    }
}
```

```

}
}

```

メソッドとクラス

	C++	Java
メソッド なしでされた	メソッド <code>virtual void eat(void) = 0;</code>	メソッド <code>abstract void draw();</code>
クラス インスタンス することはで きません	インスタンスすることはできません。 なくとも1つのメソッドをつ <code>class AB {public: virtual void f() = 0;};</code>	インスタンスすることはできません。 メソッドをつことができる <code>abstract class GraphicObject {}</code>
インタフェー ス インスタ フィールドな し	"interface"キーワードはありませんが、 クラスのをつJavaインタフェースを することができます	クラスににっていますが、1をサポート しています。2インスタンスフ ィールドなし <code>interface TestInterface {}</code>

アクセシビリティ

モディファイア	C++	Java
- すべてのユーザー がアクセス	なメモはありません	なメモはありません
された - サブクラス によってアクセス	からもアクセス	じパッケージでアクセス
- メンバーがアクセ ス	からもアクセス	なメモはありません
デフォルト	クラスのデフォルトはprivateです。のデ フォルトはpublicです。	じパッケージのすべての クラスからアクセス
その	フレンド - のないプライベートおよびさ れたメンバーへのアクセスをする	

C++のの


```
class Node {
private:
    int key;  Node *next;
    // LinkedList::search() can access "key" & "next"
    friend int LinkedList::search();
};
```

ろしいダイヤモンドの

ダイヤモンドのは、2つのクラスBとCがAからし、クラスDがBとCのからするとき
 じるあいまいさです。AにBとCがオーバーライドされ、Dがそれをオーバーライドし
 ないメソッドがある、どのバージョンのメソッドにDがしますかBのそれ、またはCの
 それはしますか [ウィキペディア](#)から

C++はにダイヤモンドのをえていましたが、JavaはJava 8までをけていましたが、Javaはのをサ
 ポートしていませんでしたが、デフォルトのインタフェースメソッドのによりJavaクラスはのク
 ラス。

java.lang.Object クラス

Javaでは、すべてのクラスはまたはにObjectクラスからされます。どのJavaもObjectにキャスト
 できます。

C++には、の「オブジェクト」クラスがありません。

JavaコレクションとC++コンテナ

JavaコレクションはC++コンテナとです。

Javaコレクションのフローチャート

C++コンテナのフローチャート

ビット			C++ LLP64またはLP64の	Javaタイプ
8	-28-1= -128	28-1-1 = 127	チャー	バイト
8	0	28-1 = 255	されていないchar	-
16	-216-1= -32,768	216-1-1 = 32,767	ショート	ショート

ビット			C ++ LLP64またはLP64の	Javaタイプ
16	0\ u0000	216-1 = 65,535\ uFFFF	のないい	charなし
32	-232-1= -2.147	232-1-1 = 21.47	int	int
32	0	232-1 = 429500	なし	-
64	-264-1	216-1-1	いです*	く い
64	0	216-1	なしlong * なしlong long	-

* Win64 APIはわずか32ビットです

よりくのC ++

Examples

クラスメンバー

メンバーはオブジェクトスコープではなくクラススコープをちます

C ++の

```
// define in header
class Singleton {
public:
    static Singleton *getInstance();

private:
    Singleton() {}
    static Singleton *instance;
};

// initialize in .cpp
Singleton* Singleton::instance = 0;
```

Javaの

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
    }
}
```

```

    }
    return instance;
}
}

```

のコンストラクトでされたクラス

のクラスでされる

C ++

ネストされたクラス[\[ref\]](#)むクラスへののです

```

class Outer {
    class Inner {
    public:
        Inner(Outer* o) :outer(o) {}

    private:
        Outer*  outer;
    };
};

```

Java

[でない]ネストされたクラスクラスまたはメンバークラス

```

class OuterClass {
    ...
    class InnerClass {
        ...
    }
}

```

のクラスでにされる

C ++

ネストされたクラス

```

class Outer {
    class Inner {
        ...
    };
};

```

Java

ネストされたクラスメンバークラス [\[ref\]](#)

```
class OuterClass {  
    ...  
    static class StaticNestedClass {  
        ...  
    }  
}
```

メソッドでされる

例えば、イベント

C ++

ローカルクラス [\[ref\]](#)

```
void fun() {  
    class Test {  
        /* members of Test class */  
    };  
}
```

Java

ローカルクラス [\[ref\]](#)

```
class Test {  
    void f() {  
        new Thread(new Runnable() {  
            public void run() {  
                doSomethingBackgroundish();  
            }  
        }).start();  
    }  
}
```

しとし

くのが、Javaはなるしだとしていますが、それよりもです。のC ++とJavaのをして、しコピーとしのくのをてください。

C ++のコード

```

// passes a COPY of the object
static void passByCopy(PassIt obj) {
    obj.i = 22; // only a "local" change
}

// passes a pointer
static void passByPointer(PassIt* ptr) {
    ptr->i = 33;
    ptr = 0; // better to use nullptr instead of '0'
}

// passes an alias (aka reference)
static void passByAlias(PassIt& ref) {
    ref.i = 44;
}

// This is an old-school way of doing it.
// Check out std::swap for the best way to do this
static void swap(PassIt** pptr1, PassIt** pptr2) {
    PassIt* tmp = *pptr1;
    *pptr1 = *pptr2;
    *pptr2 = tmp;
}

```

Javaのなコード

```

// passes a copy of the variable
// NOTE: in java only primitives are pass-by-copy
public static void passByCopy(int copy) {
    copy = 33; // only a "local" change
}

// No such thing as pointers in Java
/*
public static void passByPointer(PassIt *ptr) {
    ptr->i = 33;
    ptr = 0; // better to use nullptr instead of '0'
}
*/

// passes an alias (aka reference)
public static void passByAlias(PassIt ref) {
    ref.i = 44;
}

// passes aliases (aka references),
// but need to do "manual", potentially expensive copies
public static void swap(PassIt ref1, PassIt ref2) {
    PassIt tmp = new PassIt(ref1);
    ref1.copy(ref2);
    ref2.copy(tmp);
}

```

と

C++とJavaはどちらもオブジェクトなので、のはにされます。

したダウンキャスト

"ダウンキャスト"をすることにしてください - ダウンキャストは、をベースクラスからサブクラスすなわち、のにキャストしています。に、 instanceofdowncastingのわりに polymorphismoverridingをします。

C ++の

```
// explicit type case required
Child *pChild = (Child *) &parent;
```

Javaの

```
if(mySubClass instanceof SubClass) {
    SubClass mySubClass = (SubClass)someBaseClass;
    mySubClass.nonInheritedMethod();
}
```

メソッドとクラス

メソッド

なしでされた

C ++

メソッド

```
virtual void eat(void) = 0;
```

Java

メソッド

```
abstract void draw();
```

クラス

インスタンスすることはできません

C ++

インスタンスすることはできません。なくとも1つのメソッドをつ

```
class AB {public: virtual void f() = 0;};
```

Java

インスタンスすることはできません。メソッドをつことができる

```
abstract class GraphicObject {}
```

インタフェース

インスタンスフィールドなし

C ++

Javaにするものはもない

Java

クラスににていますが、1をサポートしています。2インスタンスフィールドなし

```
interface TestInterface {}
```

オンラインでC ++のをむ <https://riptutorial.com/ja/java/topic/10849/c-plusplus>の

8: EnumSet クラス

き

Java EnumSet クラスは、enum であるための Set です。 AbstractSet クラスを継承し、Set インタフェースを実装します。

Examples

Enum セットの

```
import java.util.*;
enum days {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
public class EnumSetExample {
    public static void main(String[] args) {
        Set<days> set = EnumSet.of(days.TUESDAY, days.WEDNESDAY);
        // Traversing elements
        Iterator<days> iter = set.iterator();
        while (iter.hasNext())
            System.out.println(iter.next());
    }
}
```

オンラインで EnumSet クラス を読む <https://riptutorial.com/ja/java/topic/10159/enumset> クラス

9: Enum マップ

き

Java EnumMap クラスは、enum キーのな Map です。 Enum および AbstractMap クラスをします。

java.util.EnumMap クラスのパラメータ

K このマップによってされるキーのタイプです。 V マップされたのタイプです。

Examples

Enum マップブックの

```
import java.util.*;
class Book {
    int id;
    String name,author,publisher;
    int quantity;
    public Book(int id, String name, String author, String publisher, int quantity) {
        this.id = id;
        this.name = name;
        this.author = author;
        this.publisher = publisher;
        this.quantity = quantity;
    }
}
public class EnumMapExample {
    // Creating enum
    public enum Key{
        One, Two, Three
    };
    public static void main(String[] args) {
        EnumMap<Key, Book> map = new EnumMap<Key, Book>(Key.class);
        // Creating Books
        Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
        Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
        Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
        // Adding Books to Map
        map.put(Key.One, b1);
        map.put(Key.Two, b2);
        map.put(Key.Three, b3);
        // Traversing EnumMap
        for(Map.Entry<Key, Book> entry:map.entrySet()){
            Book b=entry.getValue();
            System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
        }
    }
}
```

オンラインでEnumマップをむ <https://riptutorial.com/ja/java/topic/10158/enumマップ>

10: Executor、ExecutorService、Threads プール

き

Javaの`Executor`インタフェースは、スレッドの、スケジューリングなどのをむ、タスクののみからのタスクのをりすをします。、にスレッドをするわりに`Executor`がされます。 `Executors`では、はプログラムのタスクポリシーをにできるようにコードをにきすはありません。

とし

- りしするタスクをスケジュールすると、される`ScheduledExecutorService`によっては、タスクのによってされないがした、タスクはそれされなくなるがあります。

`ScheduledExecutorService`をしてください。

Examples

とれる。なタスク

エグゼキュータは、のスレッドでされるにの、またはされている、またはされているコードをむ `java.lang Runnable` をけれ `java.lang Runnable`。

はのとおりです。

```
Executor exec = anExecutor;
exec.execute(new Runnable() {
    @Override public void run() {
        //offloaded work, no need to get result back
    }
});
```

このエグゼキュータでは、されたをすがないことにしてください。

Java 8では、`lambda`をしてコードをすることができます。

Java SE 8

```
Executor exec = anExecutor;
exec.execute(() -> {
    //offloaded work, no need to get result back
});
```

ThreadPoolExecutor

されるの`Executor`は`ThreadPoolExecutor`、これはスレッドのをします。するがいときには、エグゼ

キューがにたなければならぬスレッドのコア・サイズとばれますと、プールがえるスレッド・サイズをできます。ワークロードがすると、PoolはMinのサイズにするまでスレッドのカウントを々にらします。

```
ThreadPoolExecutor pool = new ThreadPoolExecutor(
    1, // keep at least one thread ready,
    // even if no Runnable are executed
    5, // at most five Runnable/Threads
    // executed in parallel
    1, TimeUnit.MINUTES, // idle Threads terminated after one
    // minute, when min Pool size exceeded
    new ArrayBlockingQueue<Runnable>(10)); // outstanding Runnable are kept here

pool.execute(new Runnable() {
    @Override public void run() {
        //code to run
    }
});
```

キューをしてThreadPoolExecutorをコンフィグレーションすると、スレッドがのみにみしいスレッドがされるため、スレッドはcorePoolSizeえません。

すべてのパラメータをつThreadPoolExecutor

```
ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime,
    TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory,
    RejectedExecutionHandler handler)
```

JavaDocから

corePoolSizeで、maximumPoolSizeよりさいスレッドがされているは、キューがいっぱいになったにのみしいスレッドがされます。

1. BlockingQueueサイズをし、メモリシナリオをすることができます。られたキュー・サイズでは、アプリケーションのパフォーマンスはしません。
2. のRejection Handlerポリシーをすることも、しいRejection Handlerポリシーをすることもできます。
 1. デフォルトのThreadPoolExecutor.AbortPolicyでは、されると、ハンドラはRejectedExecutionExceptionをスローします。
 2. ThreadPoolExecutor.CallerRunsPolicyでは、executeをびすスレッドがタスクをします。これは、しいタスクがされるをくするなフィードバックメカニズムをする。
 3. ThreadPoolExecutor.DiscardPolicyでは、できないタスクはにされます。
 4. ThreadPoolExecutor.DiscardOldestPolicyでは、エグゼキュータがシャットダウンされてない、キューのにあるタスクがされ、がされますこれがするがあります。

3. カスタムThreadFactoryをすることができます。これはです

1. よりわかりやすいスレッドをするには
2. スレッドデーモンのステータスをするには
3. スレッドのをするには

ここでするのがあるThreadPoolExecutor

からのの。びし

がでとなるりをする、なRunnableタスクではです。このような、がしたにをす

ExecutorService.submit(Callable <T>) をできます。

サービスは、タスクのをするためにできるFutureをします。

```
// Submit a callable for execution
ExecutorService pool = anExecutorService;
Future<Integer> future = pool.submit(new Callable<Integer>() {
    @Override public Integer call() {
        //do some computation
        return new Random().nextInt();
    }
});
// ... perform other tasks while future is executed in a different thread
```

のをるがあるときは、 future.get() びします。

- のためににって、をつ。

```
try {
    // Blocks current thread until future is completed
    Integer result = future.get();
} catch (InterruptedException || ExecutionException e) {
    // handle appropriately
}
```

- がするのをちますが、されたをえてはなりません。

```
try {
    // Blocks current thread for a maximum of 500 milliseconds.
    // If the future finishes before that, result is returned,
    // otherwise TimeoutException is thrown.
    Integer result = future.get(500, TimeUnit.MILLISECONDS);
} catch (InterruptedException || ExecutionException || TimeoutException e) {
    // handle appropriately
}
```

スケジュールされたタスクまたはのタスクのがになったは、 Future.cancel(boolean) をびしてキャンセルすることができます。

- cancel(false) をびすと、するタスクのキューからタスクがされます。
- cancel(true)

をびすと、のタスクもされます。

またはりしの、にタスクをするようにスケジューリングする

`ScheduledExecutorService` クラスは、さまざまなでタスクまたはりしタスクのスケジューリングをうメソッドをします。のコードサンプルでは、`pool` がされ、のようにされているとしています。

```
ScheduledExecutorService pool = Executors.newScheduledThreadPool(2);
```

の `ExecutorService` メソッドにえて、`ScheduledExecutorService` API には、タスクを `ScheduledFuture` オブジェクトをす4つのメソッドがさ `ScheduledFuture` ます。をしてをしたり、タスクをキャンセルしたりすることができます。

のにタスクをする

のでは、10にタスクをするようにスケジュールをしています。

```
ScheduledFuture<Integer> future = pool.schedule(new Callable<>() {
    @Override public Integer call() {
        // do something
        return 42;
    }
},
10, TimeUnit.MINUTES);
```

のでタスクをする

のでは、10にタスクをし、1に1のでタスクをりしするようにスケジュールしています。

```
ScheduledFuture<?> future = pool.scheduleAtFixedRate(new Runnable() {
    @Override public void run() {
        // do something
    }
},
10, 1, TimeUnit.MINUTES);
```

タスクのは、`pool` がシャットダウンされたり、`future` がキャンセルされたり、タスクの1つがに
するまでスケジュールにつてされます。

`scheduledAtFixedRate` れた `scheduledAtFixedRate` コールによってスケジュールされたタスクは、にし
ないことがされています。タスクがのよりくかかると、のタスクのがくすることがあります。

によるタスクの

のでは、10にタスクをし、1つのタスクがしてからタスクがするまでに1のをりしてスケジュー
ルします。

```
ScheduledFuture<?> future = pool.scheduleWithFixedDelay(new Runnable() {
    @Override public void run() {
        // do something
    }
},
10, 1, TimeUnit.MINUTES);
```

タスクのは、`pool`がシャットダウンされたり、`future`がキャンセルされたり、タスクの1つがに
するまでスケジュールによってされます。

されたをする

もし

1. シャットダウンExecutorにタスクをサブミットしようとするか、または
2. キューはしていますされたスレッドでのみです、スレッドのにしました。

`RejectedExecutionHandler.rejectedExecution(Runnable, ThreadPoolExecutor)`がびされます。

デフォルトのでは、びしで`RejectedExecutionException`がスローされます。しかし、よりくのさ
れたがです。

- **ThreadPoolExecutor.AbortPolicy** デフォルト、REEをスローします
- **ThreadPoolExecutor.CallerRunsPolicy** びしのスレッドでタスクをしてスレッドをブロッ
クします
- **ThreadPoolExecutor.DiscardPolicy** タスクをサイレントに
- **ThreadPoolExecutor.DiscardOldestPolicy** キューのもいタスクをかにし、しいタスクのを
する

ThreadPool **コンストラクタ**の1つをしてそれらをすることができます

```
public ThreadPoolExecutor(int corePoolSize,
    int maximumPoolSize,
    long keepAliveTime,
    TimeUnit unit,
    BlockingQueue<Runnable> workQueue,
    RejectedExecutionHandler handler) // <--

public ThreadPoolExecutor(int corePoolSize,
    int maximumPoolSize,
    long keepAliveTime,
    TimeUnit unit,
    BlockingQueue<Runnable> workQueue,
    ThreadFactory threadFactory,
    RejectedExecutionHandler handler) // <--
```

RejectedExecutionHandlerインターフェイスをすることで、のをすることもできます。

```
void rejectedExecution(Runnable r, ThreadPoolExecutor executor)
```

submitとexecuteのいい

に、executeコマンドはfireびしとforgetびしをするなしにされ、submitコマンドはfutureオブジェクトののにされます。

これらの2つのコマンドののメカニズムのいいをするがあります。

あなたがそれらをキャッチしなかった、submitのはフレームワークによってみられます。

いをするためのコード

ケース1をするexecuteコマンドでRunnableをします。

```
import java.util.concurrent.*;
import java.util.*;

public class ExecuteSubmitDemo {
    public ExecuteSubmitDemo() {
        System.out.println("creating service");
        ExecutorService service = Executors.newFixedThreadPool(2);
        //ExtendedExecutor service = new ExtendedExecutor();
        for (int i = 0; i < 2; i++){
            service.execute(new Runnable(){
                public void run(){
                    int a = 4, b = 0;
                    System.out.println("a and b=" + a + ":" + b);
                    System.out.println("a/b:" + (a / b));
                    System.out.println("Thread Name in Runnable after divide by
zero:"+Thread.currentThread().getName());
                }
            });
        }
        service.shutdown();
    }
    public static void main(String args[]){
        ExecuteSubmitDemo demo = new ExecuteSubmitDemo();
    }
}

class ExtendedExecutor extends ThreadPoolExecutor {

    public ExtendedExecutor() {
        super(1, 1, 60, TimeUnit.SECONDS, new ArrayBlockingQueue<Runnable>(100));
    }
    // ...
    protected void afterExecute(Runnable r, Throwable t) {
        super.afterExecute(r, t);
        if (t == null && r instanceof Future<?>) {
            try {
                Object result = ((Future<?>) r).get();
            } catch (CancellationException ce) {
                t = ce;
            } catch (ExecutionException ee) {
                t = ee.getCause();
            } catch (InterruptedException ie) {
                Thread.currentThread().interrupt(); // ignore/reset
            }
        }
    }
}
```

```

    }
    if (t != null)
        System.out.println(t);
    }
}

```

```

creating service
a and b=4:0
a and b=4:0
Exception in thread "pool-1-thread-1" Exception in thread "pool-1-thread-2"
java.lang.ArithmeticException: / by zero
    at ExecuteSubmitDemo$1.run(ExecuteSubmitDemo.java:15)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
    at java.lang.Thread.run(Thread.java:744)
java.lang.ArithmeticException: / by zero
    at ExecuteSubmitDemo$1.run(ExecuteSubmitDemo.java:15)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
    at java.lang.Thread.run(Thread.java:744)

```

ケース2 **execute**を**submit**にきえます `service.submit(new Runnable() {` この、はrunメソッドがにキャッチしなかったためフレームワークによってりまれます。

```

creating service
a and b=4:0
a and b=4:0

```

ケース3 **newFixedThreadPool**を**ExtendedExecutor**にする

```

//ExecutorService service = Executors.newFixedThreadPool(2);
ExtendedExecutor service = new ExtendedExecutor();

```

```

creating service
a and b=4:0
java.lang.ArithmeticException: / by zero
a and b=4:0
java.lang.ArithmeticException: / by zero

```

このでは、2つのトピックについてしました。カスタムThreadPoolExecutorをし、カスタムThreadPoolExecutorでExceptionをします。

のにするのなのExecutorServiceおよびsubmitコマンドをしているは、FutureのsubmitコマンドおよびgetAPIからFutureオブジェクトをします。afterExecuteメソッドのでされている3つのをキャッチします。このアプローチにするカスタムThreadPoolExecutorのメカニズムは、カスタムThreadPoolExecutorという1つのでのみするがあります。

さまざまなもののユースケース

1. ExecutorService

```

ExecutorService executor = Executors.newFixedThreadPool(50);

```


シンプルでいやすいです。 `ThreadPoolExecutor` レベルのをします。

`Callable/Runnable` タスクのがなく、キューのタスクをみねてもメモリがえず、システムのパフォーマンスがするは、このをおめします。 `CPU/Memory` があるは、タスクの `RejectedExecutionHandler` をするために、 `RejectedExecutionHandler` をして `ThreadPoolExecutor` をすることをおめします。

2. `CountDownLatch`

`CountDownLatch` は、されたカウントでされます。このカウントは、 `countDown()` メソッドのびしによってされます。このカウントがゼロになるのをっているスレッドは、 `await()` メソッドの1つをびすことができます。 `await()` びすと、カウントがゼロになるまでスレッドがブロックされます。このクラスは、のスレッドセットがタスクをするまでJavaスレッドをさせます。

ユースケース

1. のをするをにするためににのスレッドをすることがあります
 2. をするにNのスレッドがするまでちます。
 3. デッドロック。
3. `ThreadPoolExecutor` よりくのをします。の `Runnable / Callable` タスクのによってアプリケーションがされているは、をしてキューをできます。キューがにすると、 `RejectionHandler` をできます。Javaには4の `RejectedExecutionHandler` **ポリシーが** `RejectedExecutionHandler` ます。
1. `ThreadPoolExecutor.AbortPolicy`、ハンドラはされるとに `RejectedExecutionException` をスローします。
 2. `ThreadPoolExecutor.CallerRunsPolicy`、`execute` をびすスレッドがタスクをします。これは、しいタスクがされるをくするなフィードバックメカニズムをする。
 3. `ThreadPoolExecutor.DiscardPolicy` では、できないタスクはにされます。
 4. `ThreadPoolExecutor.DiscardOldestPolicy`、エグゼキュータがシャットダウンされていないは、キューのにあるタスクがされ、がされますがするがあります。

`CountDownLatch` をシミュレートするは、 `invokeAll()` メソッドをできます。

4. あなたがしなかったもうつのメカニズムは `ForkJoinPool` です

`ForkJoinPool` はJava 7のJavaにされました `ForkJoinPool` はJava `ExecutorService` していますが、1つのいがあります。 `ForkJoinPool` すると、タスクをのをしてさなタスクにし、 `ForkJoinPool` することがに `ForkJoinPool` ます。きワーカースレッドがビジーのワーカースレッドキューからタスクをむとき、 `ForkJoinPool` タスクがみされます。

Java 8は、 **Work stealing pool**をするために**ExecutorService**にもう1つのAPIをしました。
`RecursiveTask`と**RecursiveAction**をするはありませんが、 `ForkJoinPool`はできます。

```
public static ExecutorService newWorkStealingPool()
```

なすべてのプロセッサをターゲットレベルとしてして、ワークスティールスレッドプールをします。

デフォルトでは、パラメータとしてCPUコアがされます。

これら4つのメカニズムはすべてにです。したいかさのレベルにじて、なものをするがあります。

ExecutorServiceのすべてのタスクのをつ

Executorにされたタスクのをつ々なオプションをてみましょう

1. **ExecutorService** `invokeAll()`

されたタスクをし、すべてがしたときにステータスとをするFuturesのリストをします。

```
import java.util.concurrent.*;
import java.util.*;

public class InvokeAllDemo{
    public InvokeAllDemo(){
        System.out.println("creating service");
        ExecutorService service =
        Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());

        List<MyCallable> futureList = new ArrayList<MyCallable>();
        for (int i = 0; i < 10; i++){
            MyCallable myCallable = new MyCallable((long)i);
            futureList.add(myCallable);
        }
        System.out.println("Start");
        try{
            List<Future<Long>> futures = service.invokeAll(futureList);
        } catch(Exception err){
            err.printStackTrace();
        }
        System.out.println("Completed");
        service.shutdown();
    }
    public static void main(String args[]){
        InvokeAllDemo demo = new InvokeAllDemo();
    }
    class MyCallable implements Callable<Long>{
        Long id = 0L;
        public MyCallable(Long val){
            this.id = val;
        }
        public Long call(){
            // Add your business logic
            return id;
        }
    }
}
```

```

    }
}
}

```

2. `CountDownLatch`

のスレッドでされているのがするまで、1つまたはのスレッドをさせる。

CountDownLatchは、されたカウントでされます。 `await`メソッドは、`countDown()`メソッドのびしによってのカウントがゼロになるまでブロックします。その、のすべてのスレッドがされ、そののすべてのwaitのびしがすぐになります。これはワンショットです。カウントをリセットすることはできません。カウントをリセットするバージョンがなは、**CyclicBarrier**のをしてください。

3. `ForkJoinPool`または`newWorkStealingPool()`でエグゼキュー

4. `ExecutorService`にしたにされたすべての`Future`オブジェクトをりしする

5. `ExecutorService`の Oracle ドキュメント・ページからのシャットダウンの

```

void shutdownAndAwaitTermination(ExecutorService pool) {
    pool.shutdown(); // Disable new tasks from being submitted
    try {
        // Wait a while for existing tasks to terminate
        if (!pool.awaitTermination(60, TimeUnit.SECONDS)) {
            pool.shutdownNow(); // Cancel currently executing tasks
            // Wait a while for tasks to respond to being cancelled
            if (!pool.awaitTermination(60, TimeUnit.SECONDS))
                System.err.println("Pool did not terminate");
        }
    } catch (InterruptedException ie) {
        // (Re-)Cancel if current thread also interrupted
        pool.shutdownNow();
        // Preserve interrupt status
        Thread.currentThread().interrupt();
    }
}

```

`shutdown()`: にサブミットされたタスクがされ、しいタスクはけられられない、なシャットダウンをします。

`shutdownNow()`: アクティブにのタスクをすべてしようとし、のタスクのをし、ちのタスクのリストをします。

のでは、タスクののがかかるは、をwhileにすることができます

```

if (!pool.awaitTermination(60, TimeUnit.SECONDS))

```

と

```

while(!pool.awaitTermination(60, TimeUnit.SECONDS)) {
    Thread.sleep(60000);
}

```

}

なるタイプの **ExecutorService** の

エグゼクティブ は、のニーズにするさまざまなタイプの **ThreadPool** をします。

1. `public static ExecutorService newSingleThreadExecutor()`

バインドされていないキューです。のワーカースレッドをする **Executor** をします。

`newFixedThreadPool(1)` と `newSingleThreadExecutor()` は、**java** ドキュメントがについてうようにいがあります。

そのの `newFixedThreadPool(1)` とはなり、されたエグゼキュータは、スレッドをするようにできないことがされています。

つまり、`newFixedThreadPool` は、`newFixedThreadPool` プログラムですることができます。

`((ThreadPoolExecutor) fixedThreadPool).setMaximumPoolSize(10)` これは `newSingleThreadExecutor` ではありません

ユースケース

1. されたタスクをにしたいとします。
2. すべてのリクエストをするスレッドは1つだけです

1. のキューはです

2. `public static ExecutorService newFixedThreadPool(int nThreads)`

キューをしてのスレッドをするスレッドプールをします。どのでも、たいていの `nThreads` スレッドはアクティブなタスクになります。すべてのスレッドがアクティブなときにのタスクがされると、スレッドがになるまでキューでします

ユースケース

1. なコアのな。 `nThreads` を `Runtime.getRuntime().availableProcessors()` としてします
`nThreads Runtime.getRuntime().availableProcessors()`
2. スレッドがスレッドプールのをえないようにすることをすると

1. のキューはです。

3. `public static ExecutorService newCachedThreadPool()`

にじてしいスレッドをするスレッドプールをします。ただし、にされたスレッドがになったにします

ユースケース

- 1.

のタスク

1. のキューはです。
2. のスレッドがすべてビジーになると、しいスレッドがされます。タスクのがくになると、よりくのスレッドがされ、システムのパフォーマンスがします。このの

`newFixedThreadPool`

4. `public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)`

されたのにするコマンド、またはにするコマンドをスケジュールできるスレッドプールをします。

ユースケース

1. をうなイベントのは、のでわれます。

1. のキューはです。

5. `public static ExecutorService newWorkStealingPool()`

なすべてのプロセッサをターゲットレベルとしてして、ワークスティールスレッドプールをします。

ユースケース

1. してするタイプの。
2. アイドルのスレッドのな。アイドルのスレッドは、ビジーのスレッドからタスクをみます。

1. のキューサイズはです。

これらのすべての`ExecutorService`には、のキューというのが1つあります。これは、[ThreadPoolExecutor](#)でされます。

```
ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime,
    TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory,
    RejectedExecutionHandler handler)
```

`ThreadPoolExecutor`、

1. スレッドプールサイズをにする
2. `BlockingQueue`のをする
3. キューがいっぱいになると`RejectedExecutionHandler`する
4. `CustomThreadFactory`をしてスレッドにをする (`public Thread newThread(Runnable r)`)

スレッドプールの

スレッドプールは、に`ExecutorService`メソッドをびすためにされます。

のメソッドをして、するをできます。

submit	されたをし、をるためにできるをす
execute	にらかのりをることなくタスクをする
invokeAll	タスクのリストをし、のリストをす
invokeAny	すべてをするが、したのみをすなし

スレッドプールをすると、`shutdown()`をびしてスレッドプールをできます。のすべてのタスクがされます。すべてのタスクがされるのをつために、`awaitTermination`または`isShutdown()`ループすることができます。

オンラインでExecutor、ExecutorService、Threadsプールをむ

<https://riptutorial.com/ja/java/topic/143/executor-executorservice-threads>プール

11: FileUpload to AWS

き

スプリングレスAPIをしてAWS s3バケットにファイルをアップロードします。

Examples

ファイルをs3バケットにアップロードする

ここでは、ファイルオブジェクトをフロントエンドからマルチパートパラメータとして送り、それをJavaレストAPIをしてS3バケットにアップロードする、りのAPIをします。

- ファイルをアップロードしたいs3バケットのとアクセスキー。

コード - DocumentController.java

```
@RestController
@RequestMapping("/api/v2")
public class DocumentController {

    private static String bucketName = "pharmerz-chat";
    // private static String keyName = "Pharmerz"+ UUID.randomUUID();

    @RequestMapping(value = "/upload", method = RequestMethod.POST, consumes =
    MediaType.MULTIPART_FORM_DATA)
    public URL uploadFileHandler(@RequestParam("name") String name,
                                @RequestParam("file") MultipartFile file) throws IOException
    {

        /***** Printing all the possible parameter from @RequestParam *****/

        System.out.println("*****");

        System.out.println("file.getOriginalFilename() " + file.getOriginalFilename());
        System.out.println("file.getContentType() " + file.getContentType());
        System.out.println("file.getInputStream() " + file.getInputStream());
        System.out.println("file.toString() " + file.toString());
        System.out.println("file.getSize() " + file.getSize());
        System.out.println("name " + name);
        System.out.println("file.getBytes() " + file.getBytes());
        System.out.println("file.hashCode() " + file.hashCode());
        System.out.println("file.getClass() " + file.getClass());
        System.out.println("file.isEmpty() " + file.isEmpty());

        /*****Parameters to b pass to s3 bucket put Object *****/
        InputStream is = file.getInputStream();
        String keyName = file.getOriginalFilename();

        // Credentials for Aws
```

```

        AWSCredentials credentials = new BasicAWSCredentials("AKIA*****",
"zr*****");

        /***** DocumentController.uploadfile(credentials);
*****/

        AmazonS3 s3client = new AmazonS3Client(credentials);
        try {
            System.out.println("Uploading a new object to S3 from a file\n");
            //File file = new File(awsuploadfile);
            s3client.putObject(new PutObjectRequest(
                bucketName, keyName, is, new ObjectMetadata()));

            URL url = s3client.generatePresignedUrl(bucketName, keyName,
Date.from(Instant.now().plus(5, ChronoUnit.MINUTES)));
            // URL url=s3client.generatePresignedUrl(bucketName,keyName,
Date.from(Instant.now().plus(5, ChronoUnit.)));
            System.out.println("*****");
            System.out.println(url);

            return url;

        } catch (AmazonServiceException ase) {
            System.out.println("Caught an AmazonServiceException, which " +
                "means your request made it " +
                "to Amazon S3, but was rejected with an error response" +
                " for some reason.");
            System.out.println("Error Message: " + ase.getMessage());
            System.out.println("HTTP Status Code: " + ase.getStatusCode());
            System.out.println("AWS Error Code: " + ase.getErrorCode());
            System.out.println("Error Type: " + ase.getErrorType());
            System.out.println("Request ID: " + ase.getRequestId());
        } catch (AmazonClientException ace) {
            System.out.println("Caught an AmazonClientException, which " +
                "means the client encountered " +
                "an internal error while trying to " +
                "communicate with S3, " +
                "such as not being able to access the network.");
            System.out.println("Error Message: " + ace.getMessage());
        }

        return null;
    }

}

```

フロントエンド

```

var form = new FormData();
form.append("file", "image.jpeg");

var settings = {
    "async": true,
    "crossDomain": true,
    "url": "http://url/",
    "method": "POST",

```



```
"headers": {
  "cache-control": "no-cache"
},
"processData": false,
"contentType": false,
"mimeType": "multipart/form-data",
"data": form
}

$.ajax(settings).done(function (response) {
  console.log(response);
});
```

オンラインでFileUpload to AWSをむ <https://riptutorial.com/ja/java/topic/10589/fileupload-to-aws>

12: FTP ファイルプロトコル

- FTPClient connectInetAddress ホスト、int ポート
- FTPClient ログイン ユーザー、パスワード
- FTPClient disconnect
- FTPReply getReplyStrings
- boolean storeFileString remote、InputStream local
- OutputStream storeFileStreamString remote
- boolean setFileTypeint fileType
- boolean completePendingCommand

パラメーター

パラメーター	
ホスト	FTPサーバーのホストまたはIPアドレス
	FTPサーバーポート
ユーザー	FTPサーバーのユーザー
パスワード	FTPサーバーのパスワード

Examples

FTPサーバーへのログイン

JavaでFTPをするには、しいFTPClientをし、.connect(String server, int port)と.login(String username, String password)をしてサーバーにしてログインするがあり.login(String username, String password)。

```
import java.io.IOException;
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPReply;
//Import all the required resource for this project.

public class FTPConnectAndLogin {
    public static void main(String[] args) {
        // SET THESE TO MATCH YOUR FTP SERVER //
        String server = "www.server.com"; //Server can be either host name or IP address.
        int port = 21;
        String user = "Username";
        String pass = "Password";

        FTPClient ftp = new FTPClient();
        ftp.connect(server, port);
```

```

        ftp.login(user, pass);
    }
}

```

これでがしました。しかし、サーバーにするにエラーがしたはどうすればよいでしょうかかがうまくいかないときをり、エラーメッセージをけることがになります。にエラーをするコードをしましょう。

```

try {
    ftp.connect(server, port);
    showServerReply(ftp);
    int replyCode = ftp.getReplyCode();
    if (!FTPReply.isPositiveCompletion(replyCode)) {
        System.out.println("Operation failed. Server reply code: " + replyCode);
        return;
    }
    ftp.login(user, pass);
} catch {

}

```

たちがやったことをみこんでみましょう。

```

showServerReply(ftp);

```

これは、のでするをします。

```

int replyCode = ftp.getReplyCode();

```

これは、サーバーからの/エラーコードをし、としてします。

```

if (!FTPReply.isPositiveCompletion(replyCode)) {
    System.out.println("Operation failed. Server reply code: " + replyCode);
    return;
}

```

これは、エラーがあったかどうかをするためにコードをチェックします。エラーがしたは、に「にしました。サーバーコード」とエラーコードがされます。またtry / catchブロックをし、のステップでします。に、 ftp.login()のエラーをチェックするをしましょう。

```

boolean success = ftp.login(user, pass);
showServerReply(ftp);
if (!success) {
    System.out.println("Failed to log into the server");
    return;
} else {
    System.out.println("LOGGED IN SERVER");
}

```

このブロックをしましょう。

```
boolean success = ftp.login(user, pass);
```

これは、FTPサーバーへのログインをみるだけでなく、をブールとしてします。

```
showServerReply(ftp);
```

これにより、サーバーからのメッセージがされたかどうかはされますが、まずのでをすることがあります。

```
if (!success) {
    System.out.println("Failed to log into the server");
    return;
} else {
    System.out.println("LOGGED IN SERVER");
}
```

このステートメントは、にログインしたかどうかをします。そうであれば、"LOGGED IN SERVER"とされます。それのは、"サーバへのログインにしました"とされます。これはこれまでのスクリプトです

```
import java.io.IOException;
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPReply;

public class FTPConnectAndLogin {
    public static void main(String[] args) {
        // SET THESE TO MATCH YOUR FTP SERVER //
        String server = "www.server.com";
        int port = 21;
        String user = "username"
        String pass = "password"

        FTPClient ftp = new FTPClient
        try {
            ftp.connect(server, port)
            showServerReply(ftp);
            int replyCode = ftpClient.getReplyCode();
            if (!FTPReply.isPositiveCompletion(replyCode)) {
                System.out.println("Operation failed. Server reply code: " + replyCode);
                return;
            }
            boolean success = ftp.login(user, pass);
            showServerReply(ftp);
            if (!success) {
                System.out.println("Failed to log into the server");
                return;
            } else {
                System.out.println("LOGGED IN SERVER");
            }
        } catch {

        }
    }
}
```

に、プロセスでエラーがしたにえて、Catchブロックをしましょう。

```
} catch (IOException ex) {
    System.out.println("Oops! Something went wrong.");
    ex.printStackTrace();
}
```

したキャッチブロックに「OopsSomething went wrong」とされるようになりました。エラーがあるはスタックトレースをします。、たちののステップは、っているshowServerReply() をすること showServerReply() 。

```
private static void showServerReply(FTPClient ftp) {
    String[] replies = ftp.getReplyStrings();
    if (replies != null && replies.length > 0) {
        for (String aReply : replies) {
            System.out.println("SERVER: " + aReply);
        }
    }
}
```

このは、 FTPClient をとしてり、それを "ftp" とびます。その、サーバからののサーバをにします。に、メッセージがされているかどうかをします。いずれかがあるは、それぞれを「SERVER [reply]」としてします。そのがしたので、これはしたスクリプトです

```
import java.io.IOException;
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPReply;

public class FTPConnectAndLogin {
    private static void showServerReply(FTPClient ftp) {
        String[] replies = ftp.getReplyStrings();
        if (replies != null && replies.length > 0) {
            for (String aReply : replies) {
                System.out.println("SERVER: " + aReply);
            }
        }
    }

    public static void main(String[] args) {
        // SET THESE TO MATCH YOUR FTP SERVER //
        String server = "www.server.com";
        int port = 21;
        String user = "username"
        String pass = "password"

        FTPClient ftp = new FTPClient
        try {
            ftp.connect(server, port)
            showServerReply(ftp);
            int replyCode = ftpClient.getReplyCode();
            if (!FTPReply.isPositiveCompletion(replyCode)) {
                System.out.println("Operation failed. Server reply code: " + replyCode);
                return;
            }
            boolean success = ftp.login(user, pass);
            showServerReply(ftp);
        }
```

```

        if (!success) {
            System.out.println("Failed to log into the server");
            return;
        } else {
            System.out.println("LOGGED IN SERVER");
        }
    } catch (IOException ex) {
        System.out.println("Oops! Something went wrong.");
        ex.printStackTrace();
    }
}
}

```

にしいFTPClientをし、それをサーバーにし、.connect(String server, int port)と.login(String username, String password)をしてログインする.login(String username, String password)。コードがサーバーにできないは、try / catchブロックをしてしてログインすることがです。また、とログインをみるときにサーバからけったメッセージをチェックしてするをするがあります。このを "showServerReply(FTPClient ftp) "とびます。

```

import java.io.IOException;
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPReply;

public class FTPConnectAndLogin {
    private static void showServerReply(FTPClient ftp) {
        if (replies != null && replies.length > 0) {
            for (String aReply : replies) {
                System.out.println("SERVER: " + aReply);
            }
        }
    }

    public static void main(String[] args) {
        // SET THESE TO MATCH YOUR FTP SERVER //
        String server = "www.server.com";
        int port = 21;
        String user = "username"
        String pass = "password"

        FTPClient ftp = new FTPClient
        try {
            ftp.connect(server, port)
            showServerReply(ftp);
            int replyCode = ftpClient.getReplyCode();
            if (!FTPReply.isPositiveCompletion(replyCode)) {
                System.out.println("Operation failed. Server reply code: " + replyCode);
                return;
            }
            boolean success = ftp.login(user, pass);
            showServerReply(ftp);
            if (!success) {
                System.out.println("Failed to log into the server");
                return;
            } else {
                System.out.println("LOGGED IN SERVER");
            }
        } catch (IOException ex) {
            System.out.println("Oops! Something went wrong.");
        }
    }
}

```

```
        ex.printStackTrace();  
    }  
}  
}
```

その、FTPサーバーをJavaスクリプトにしてください。

オンラインでFTPファイルプロトコルをむ <https://riptutorial.com/ja/java/topic/5228/ftp-ファイルプロトコル->

13: HttpURLConnection

- AndroidでHttpURLConnectionをするには、AndroidManifest.xmlのアプリにインターネットアクセスをするがあります。
- SquareのOkHttpなど、のJava HTTPクライアントとライブラリもいやすく、よりれたパフォーマンスやをすることができます。

Examples

レスポンスをURLからStringとしてする

```
String getText(String url) throws IOException {
    HttpURLConnection connection = (HttpURLConnection) new URL(url).openConnection();
    //add headers to the connection, or check the status if desired..

    // handle error response code it occurs
    int responseCode = conn.getResponseCode();
    InputStream inputStream;
    if (200 <= responseCode && responseCode <= 299) {
        inputStream = connection.getInputStream();
    } else {
        inputStream = connection.getErrorStream();
    }

    BufferedReader in = new BufferedReader(
        new InputStreamReader(
            inputStream));

    StringBuilder response = new StringBuilder();
    String currentLine;

    while ((currentLine = in.readLine()) != null)
        response.append(currentLine);

    in.close();

    return response.toString();
}
```

これにより、されたURLからテキストデータがダウンロードされ、Stringとしてされます。

このみ

- まず、new URL(url).openConnection()をして、URLからHttpURLConnectionをします。
URLConnectionをキャストしてURLConnectionすので、ヘッダーUser AgentなどのやレスポンスコードなどのHttpURLConnectionにアクセスできます。 このではそうしませんが、はです。
- に、レスポンスコードについてInputStreamしますエラー
- に、からしたInputStreamからテキストをみることができるBufferedReaderをします。

- は、テキストを `StringBuilder` します。
- `InputStream` じ、っているをします。

ノート

- このメソッドは、ネットワークエラーやインターネットなどがしたに `IOException` をスローし、`IOException` されたURLがでないはチェックされていない `MalformedURLException` もスローします。
- これは、WebページHTML、JSONやXMLをすREST APIなど、テキストをすのURLからみるためにできます。
- [のJavaコードでURLをStringにみみます](#)。

にです

```
String text = getText("http://example.com");
//Do something with the text from example.com, in this case the HTML.
```

POSTデータ

```
public static void post(String url, byte [] data, String contentType) throws IOException {
    HttpURLConnection connection = null;
    OutputStream out = null;
    InputStream in = null;

    try {
        connection = (HttpURLConnection) new URL(url).openConnection();
        connection.setRequestProperty("Content-Type", contentType);
        connection.setDoOutput(true);

        out = connection.getOutputStream();
        out.write(data);
        out.close();

        in = connection.getInputStream();
        BufferedReader reader = new BufferedReader(new InputStreamReader(in));
        String line = null;
        while ((line = reader.readLine()) != null) {
            System.out.println(line);
        }
        in.close();

    } finally {
        if (connection != null) connection.disconnect();
        if (out != null) out.close();
        if (in != null) in.close();
    }
}
```

これにより、されたURLにPOSTデータがされ、が1ずつみまれます。

い

- どり、 URL から `URLConnection` をし URL 。
- `setRequestProperty` をしてコンテンツタイプをし `setRequestProperty` 。 デフォルトでは `application/x-www-form-urlencoded`
- `setDoOutput(true)` は、 データをすることをにします。
- に、 `getOutputStream()` をびして `OutputStream` をし、 それにデータをきみます。 にそれをする ことをれないでください。
- に、 サーバーのをみました。

リソースをする

```
public static void delete (String urlString, String contentType) throws IOException {
    HttpURLConnection connection = null;

    try {
        URL url = new URL(urlString);
        connection = (HttpURLConnection) url.openConnection();
        connection.setDoInput(true);
        connection.setRequestMethod("DELETE");
        connection.setRequestProperty("Content-Type", contentType);

        Map<String, List<String>> map = connection.getHeaderFields();
        StringBuilder sb = new StringBuilder();
        Iterator<Map.Entry<String, String>> iterator =
responseHeader.entrySet().iterator();
        while(iterator.hasNext())
        {
            Map.Entry<String, String> entry = iterator.next();
            sb.append(entry.getKey());
            sb.append('=').append('"');
            sb.append(entry.getValue());
            sb.append('"');
            if(iterator.hasNext())
            {
                sb.append(',').append(' ');
            }
        }
        System.out.println(sb.toString());

    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (connection != null) connection.disconnect();
    }
}
```

これにより、されたURLのリソースがされ、レスポンスヘッダーがされます。

い

- URL から `URLConnection` をし URL 。

- `setRequestProperty`をしてコンテンツタイプをし`setRequestProperty`。デフォルトでは `application/x-www-form-urlencoded`
- `setDoInput(true)`は、にURLをすることをにします。
- HTTP DELETEをするための`setRequestMethod("DELETE")`

に、サーバーレスポンスヘッダーをします。

リソースがするかどうかをする

```
/**
 * Checks if a resource exists by sending a HEAD-Request.
 * @param url The url of a resource which has to be checked.
 * @return true if the response code is 200 OK.
 */
public static final boolean checkIfResourceExists(URL url) throws IOException {
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setRequestMethod("HEAD");
    int code = conn.getResponseCode();
    conn.disconnect();
    return code == 200;
}
```

リソースがするかどうかだけをしているは、GETよりもHEADリクエストをするがよいでしょう。これにより、リソースをするオーバーヘッドがされます。

レスポンスコードが200にのみ、このメソッドは`true`し`true`。リダイレクトつまり3XXのをしている、そのメソッドをするがあります。

```
checkIfResourceExists(new URL("http://images.google.com/")); // true
checkIfResourceExists(new URL("http://pictures.google.com/")); // false
```

オンラインでHttpURLConnectionをむ <https://riptutorial.com/ja/java/topic/156/httpurlconnection>

14: InputStreams と OutputStreams

- `int readbyte []` bthrows `IOException`

ほとんどの、`InputStream`せず、`BufferedStream`することにしてください。これは、`InputStream`が`read`メソッドがびされるたびにソースからみむためです。これは、カーネルとのでのコンテキストスイッチのなCPUをきこすがあります。

Examples

InputStreamをにみむ

によっては、バイトをStringにみむこともできます。これをうには、`byte`と「ネイティブJava」UTF-16コードポイントのをするものを`char`としてするがあります。これは`InputStreamReader`ます。

プロセスをしするため、バッファをりてるのはですが、Inputからみむにオーバーヘッドがあまりありません。

Java SE 7

```
public String inputStreamToString(InputStream inputStream) throws Exception {
    StringWriter writer = new StringWriter();

    char[] buffer = new char[1024];
    try (Reader reader = new BufferedReader(new InputStreamReader(inputStream, "UTF-8"))) {
        int n;
        while ((n = reader.read(buffer)) != -1) {
            // all this code does is redirect the output of `reader` to `writer` in
            // 1024 byte chunks
            writer.write(buffer, 0, n);
        }
    }
    return writer.toString();
}
```

これをJava SE 6およびそれコードにすることは、のとしてはされています。

OutputStreamへのバイトのきみ

に1バイトずつOutputStreamバイトをきむ

```
OutputStream stream = object.getOutputStream();

byte b = 0x00;
stream.write( b );
```

バイトのきみ

```
byte[] bytes = new byte[] { 0x00, 0x00 };

stream.write( bytes );
```

バイトのセクションをく

```
int offset = 1;
int length = 2;
byte[] bytes = new byte[] { 0xFF, 0x00, 0x00, 0xFF };

stream.write( bytes, offset, length );
```

ストリームをじる

ほとんどのストリームは、したでじなければなりません。そうしないと、メモリリークがしたり、ファイルをいたままにしたりするがあります。がスローされてもストリームがじられることがです。

Java SE 7

```
try(FileWriter fw = new FileWriter("outfilename");
    BufferedWriter bw = new BufferedWriter(fw);
    PrintWriter out = new PrintWriter(bw))
{
    out.println("the text");
    //more code
    out.println("more text");
    //more code
} catch (IOException e) {
    //handle this however you
}
```

えておいてくださいtry-with-resourcesは、ブロックがしたときにリソースがじられたこと、のフローでしたのかどうか、またはのためにリソースがしたことをします。

Java SE 6

リソースをそうとするのはオプションではないもあれば、いバージョンのJava 6をサポートしていることもあります。この、なはfinallyブロックをすることです。

```
FileWriter fw = null;
BufferedWriter bw = null;
PrintWriter out = null;
try {
    fw = new FileWriter("myfile.txt");
    bw = new BufferedWriter(fw);
    out = new PrintWriter(bw);
    out.println("the text");
    out.close();
} catch (IOException e) {
    //handle this however you want
}
finally {
```

```
try {
    if(out != null)
        out.close();
} catch (IOException e) {
    //typically not much you can do here...
}
}
```

ラッパーストリームをじると、そののストリームもじられることにしてください。つまり、ストリームをラップすることはできず、ラッパーをじてからのストリームをしけることができます。

ストリームをストリームにコピーする

これは、2つのストリームでデータをコピーします。

```
void copy(InputStream in, OutputStream out) throws IOException {
    byte[] buffer = new byte[8192];
    while ((bytesRead = in.read(buffer)) > 0) {
        out.write(buffer, 0, bytesRead);
    }
}
```

-

```
// reading from System.in and writing to System.out
copy(System.in, System.out);
```

ストリームとストリームのりし

`OutputStream`と`InputStream`はくのかなるクラスがあり、それぞれのをっています。ストリームをのストリームにラップすることで、のストリームのがられます。

でもストリームをラップすることができます。をきめてください。

なみわせ

バッファをしているときにファイルにをきむ

```
File myFile = new File("targetFile.txt");
PrintWriter writer = new PrintWriter(new BufferedOutputStream(new FileOutputStream(myFile)));
```

バッファをしてファイルにきむにデータをおよびする

```
Cipher cipher = ... // Initialize cipher
File myFile = new File("targetFile.enc");
BufferedOutputStream outputStream = new BufferedOutputStream(new DeflaterOutputStream(new
CipherOutputStream(new FileOutputStream(myFile), cipher)));
```

ストリームラッパーのリスト

ラッパー	
BufferedOutputStream / BufferedInputStream	OutputStreamはデータを1バイトずつきみますが、BufferedOutputStreamはデータをチャンクにきみます。これにより、システムコールのがり、パフォーマンスがします。
DeflaterOutputStream / DeflaterInputStream	データをします。
InflaterOutputStream / InflaterInputStream	データののをします。
CipherOutputStream / CipherInputStream	データを/します。
DigestOutputStream / DigestInputStream	メッセージのダイジェストをしてデータののをします。
ChecksumOutputStream / ChecksumInputStream	Checksumをします。ChecksumはMessage Digestのよりなバージョンです。
DataOutputStream / DataInputStream	プリミティブデータとのきみをします。バイトをくための。プラットフォームにしない。
PrintStream	プリミティブデータとのきみをします。バイトをくための。プラットフォームに。
OutputStreamWriter	OutputStreamをライターにします。Writerがをする、OutputStreamはバイトをいます
PrintWriter	OutputStreamWriterをにびします。プリミティブデータとのきみをします。にをくため、そしてをくのにです

DataInputStreamの

```
package com.streams;
import java.io.*;
public class DataStreamDemo {
    public static void main(String[] args) throws IOException {
        InputStream input = new FileInputStream("D:\\datastreamdemo.txt");
        DataInputStream inst = new DataInputStream(input);
        int count = input.available();
        byte[] arr = new byte[count];
        inst.read(arr);
        for (byte byt : arr) {
```

```
        char ki = (char) byt;  
        System.out.print(ki+"-");  
    }  
}  
}
```

オンラインでInputStreamsとOutputStreamsをむ

<https://riptutorial.com/ja/java/topic/110/inputstreamsとoutputstreams>

15: Java Edition、バージョン、リリースおよびディストリビューション

Examples

Java SE JRE または Java SE JDK ディストリビューションのい

Sun / Oracle リリースの Java SE には、JRE と JDK という 2 つのがあります。にえば、JRE はの Java アプリケーションをサポートし、JDK は Java もサポートします。

Java ランタイム

Java Runtime Environment または JRE ディストリビューションは、Java アプリケーションのとな のライブラリとツールでされています。な JRE のツールはのとおりです。

- JVM Java Virtual Machine で Java プログラムをするための `java` コマンドは、
- Nashorn Javascript エンジンをするための `js` コマンド。
- Java キーストアをするための `keytool` コマンド。
- セキュリティサンドボックスのセキュリティポリシーをするための `policytool` コマンド。
- Web の「pack200」ファイルのとのための `pack200` と `unpack200` ツール。
- Java CORBA および RMI アプリケーションをサポートする `orbd`、`rmid`、`rmiregistry` および `tnameserv` コマンド。

「デスクトップ JRE」インストーラには、の Web ブラウザにした Java プラグインがまれています。これはに「Server JRE」インストーラーからされています。linux syscall read benchmark

Java 7 Update 6 では、JRE インストーラーに JavaFX バージョン 2.2 がまれています。

Java キット

Java Development Kit または JDK ディストリビューションには、JRE ツール、および Java ソフトウェアをするためのツールがまれています。のツールには、

- `javac` コマンド。Java ソースコード `".java"` をバイトコードファイル `".class"` にコンパイルします。
- `jar` や `jarsigner` などの JAR ファイルをするためのツール
- のようなツール
 - `appletviewer` アプレットをするための
 - `idlj` CORBA IDL から Java コンパイラへ
 - `javah` JNI スタブジェネレータ
 - Java ソースコードのセットの `native2ascii`

- `schemagen` Java to XMLスキーマジェネレータJAXBの
- `serialver`は、Java Object Serializationバージョンをします。
- JAX-WSのための`wsgen`と`wsimport` サポートツール
- ツール
 - `jdb`なJavaデバッガ
 - `jmap`および`jhat`をしてJavaヒープをダンプおよびします。
 - スレッドスタックダンプをするための`jstack`
 - ".class"ファイルをべるための`javap`
- アプリケーションおよびツール
 - `jconsole`はコンソール、
 - アプリケーションの`jstat`、`jstatd`、`jinfo`、および`jps`

なSun / Oracle JDKインストールには、JavaライブラリのソースコードをむZIPファイルもまれています。Java 6よりは、されているのJavaソースコードでした。

Java 6、OpenJDKのなソースコードはOpenJDKサイトからダウンロードできます。、LinuxJDKパッケージにはまれていませんが、のパッケージとしてできます。

Oracle HotspotとOpenJDKのいはですか

JREとJDKのとしているため、くできるJavaリリースには2あります。

- Oracleホットスポットのリリースは、Oracleのダウンロードサイトからダウンロードしたものです。
- OpenJDKのリリースは、OpenJDKのソースリポジトリからされたリリースはサードパーティのプロバイダです。

には、HotspotリリースとOpenJDKリリースにはほとんどいがありません。ホットスポットには、OracleJavaのがにできる「エンタープライズ」がいくつかありますが、ホットスポットとOpenJDKのにじテクノロジーがします。

OpenJDKにするHotspotのもう1つののは、Hotspotのパッチリリースがしにリリースされるがあることです。これは、OpenJDKプロバイダがどのであるかによってもなります。たとえば、LinuxディストリビューションのビルドチームがしいOpenJDKビルドをしてQAをし、パブリックリポジトリにれるのにどれくらいのがかかるかなどです。

のホットスポットリリースは、ほとんどのLinuxディストリビューションのパッケージリポジトリからはできません。これは、Hotspotをすると、LinuxマシンでJavaソフトウェアをのにつことがよりであることをします。

Java EE、Java SE、Java ME、およびJavaFXのい

Javaテクノロジーはプログラミングとプラットフォームのです。Javaプログラミングは、のとスタイルをつのオブジェクトです。Javaプラットフォームは、Javaプログラミングアプリケーションがされるのです。

いくつかのJavaプラットフォームがあります。くのは、のJavaプログラミングであっても、なるプラットフォームがいにとどのようにしているかをしていません。

Javaプログラミングプラットフォーム

Javaプログラミングには4つのプラットフォームがあります。

- Javaプラットフォーム、Standard EditionJava SE
- Javaプラットフォーム、エンタープライズJava EE
- Javaプラットフォーム、Micro EditionJava ME
- Java FX

すべてのJavaプラットフォームは、JavaマシンVMとアプリケーションプログラミングインターフェイスAPIでされています。Java Virtual Machineは、Javaテクノロジアプリケーションをするのハードウェアおよびソフトウェアプラットフォームのプログラムです。APIは、のソフトウェアコンポーネントまたはアプリケーションをするためにできるソフトウェアコンポーネントのまりです。JavaプラットフォームはマシンとAPIをします。これにより、Javaプラットフォームの、、、の、およびセキュリティ。

Java SE

ほとんどのがJavaプログラミングをえるとき、らはJava SE APIをえています。Java SEのAPIは、Javaプログラミングのコアをします。これは、Javaプログラミングのタイプとオブジェクトから、ネットワーク、セキュリティ、データベースアクセス、グラフィカルユーザーインターフェイスGUI、およびXMLにされるクラスまでをすべてします。

コアAPIにえて、Java SEプラットフォームは、マシン、ツール、デプロイメントテクノロジ、およびJavaテクノロジアプリケーションでにされるそののクラスライブラリとツールキットでされています。

Java EE

Java EEプラットフォームは、Java SEプラットフォームのにされています。Java EEプラットフォームは、でスケラブルでなく、なネットワークアプリケーションをしするためのAPIとをします。

Java ME

Java MEプラットフォームは、などのデバイスでJavaプログラミングアプリケーションをするためのAPIと小さなフットプリントマシンをします。このAPIはJava SE APIのサブセットであり、小さなデバイスアプリケーションのにつなクラスライブラリもあります。Java MEアプリケーションは、くの、Java EEプラットフォームサービスのクライアントです。

Java FX

Java FXテクノロジーは、Java FX Script™でかれたリッチインターネットアプリケーションをするためのプラットフォームです。Java FXスクリプトは、Javaテクノロジーバイトコードにコンパイルされ、Java VMでされます。Java FXプラットフォームにされたアプリケーションは、Javaプログラミングクラスをみ、それにリンクすることができ、Java EEプラットフォームサービスのクライアントでもよい。

- [Oracleのドキュメント](#)から

Java SEのバージョン

Java SEのバージョン

のは、Java SEプラットフォームのなメジャーバージョンのタイムラインをしています。

Java SEバージョン ¹	コード	2	
Java SE 9アーリーアクセス	し		2017-07-27
Java SE 8	し		2014-03-18
Java SE 7	イルカ	2015-04-14	2011-07-28
Java SE 6	マスタング	2013-04-16	2006-12-23
Java SE 5		2009-11-04	2004104
Java SE 1.4.2	マンティス	2009114より	2003-06-26
Java SE 1.4.1	ホッパー/グラスホッパー	2009114より	2002916
Java SE 1.4	マーリン	2009114より	2002-02-06
Java SE 1.3.1	レディバード	2009114より	2001-05-17
Java SE 1.3	チョウゲンボウ	2009114より	2000-05-08
Java SE 1.2	び	2009114より	1998128

Java SEバージョン ¹	コード	2	
Java SE 1.1	スパークラー	2009114より	1997-02-19
Java SE 1.0	オーク	2009114より	1996121

1. このリンクは、オラクルのWebサイトにあるリリースのドキュメントのオンライン・コピーへのリンクです。このリリースのドキュメントはオンラインではなくなりましたが、はOracle Javaアーカイブからダウンロードできます。
2. Java SEのほとんどのバージョンはな "end of life"のをぎました。Javaバージョンがこのマイルストーンにすると、Oracleはでアップデートをしなくなります。サポートは、サポートをんでいるおにもできます。

ソース

- カナダのマインド・プロダクツのRoedy Greenによる[JDKリリース](#)

Java SE バージョンのハイライト

Java SE バージ ョン	ハイライト
Java SE 8	ラムダとMapReduceにインスパイアされたストリーム。 Nashorn Javascriptエンジン。 のとりしの。 なし。 しいとのAPI。 にリンクされたJNIライブラリ。 JavaFXランチャー。 PermGenの。
Java SE 7	スイッチ、 リソースの、 ダイヤモンド <>、 リテラルの、 /のライブラリの。 ネイティブファイルシステムのサポートがされました。 ティムソート。 ECCアルゴリズム。 された2DグラフィックスGPUサポート。 プラuggブルな。
Java SE 6	JVMプラットフォームとSwingのパフォーマンスのなスクリプトAPIとMozilla Rhino Javascriptエンジン。 JDBC 4.0。 コンパイラAPI。 JAXB 2.0。 WebサービスのサポートJAX-WS
Java SE 5	、 ボクシング、 enumクラス、 varargs、 forループとなインポートのfor。 Javaメモリモデルの。 スィングとRMIの。 java.util.concurrent.*パッケージとScanner。
Java SE 1.4	assertキーワード。 クラス。 チェーン。 NIO API - ノンブロッキングI/O、 Buffer、 Channel。 java.util.logging.* API。 イメージI/O API。 されたXMLとXSLT JAXP。 されたセキュリティとJCE、 JSSE、 JAASされたJava Web Start。 プリファレンスAPI。
Java SE 1.3	HotSpot JVMがまれています。 CORBA / RMIJavaとディレクトリインタフェースJNDI。 デバッガフレームワークJPDA。 JavaSound API。 プロキシAPI。

Java SE バージョン	ハイライト
Java SE 1.2	<code>strictfp</code> キーワード。 Swing API。 JavaプラグインWebブラウザ。 CORBAのコレクションフレームワーク。
Java SE 1.1	クラス。。 JDBC。 RMI。 Unicode /ストリーム。。 AWTイベントモデルのオーバーホールJavaBeans。

ソース

- Wikipedia [Javaのバージョン](#)

オンラインでJava Edition、バージョン、リリースおよびディストリビューションをむ
<https://riptutorial.com/ja/java/topic/8973/java-edition-バージョン-リリースおよびディストリビューション>

16: Java SE 7の

き

このトピックでは、Java SE 7のJavaプログラミングにされたのをします。JDBCやJava Virtual MachineJVMなど、のには、このトピックでは

Java SE 7の

Examples

しいJava SE 7プログラミング

- **バイナリリテラル** byte、short、int、longは、バイナリナンバーシステムをしてすることもできます。バイナリリテラルをするには、にプレフィックス0bまたは0Bをします。
- **switchの switch**のにStringオブジェクトをできます
- **try-with-resourcesステートメント** try-with-resourcesステートメントは、1つのリソースをするtryステートメントです。リソースは、プログラムがしたでじなければならぬオブジェクトとしてします。 try-with-resourcesステートメントは、リソースがステートメントのにじられるようにします。 java.lang.AutoCloseableをするオブジェクトは、java.io.Closeableをするすべてのオブジェクトをみ、リソースとしてできます。
- **されたチェックによるののキャッチとの**のキャッチブロックは、ののをできます。このは、コードのをらし、にいをキャッチするをすることができます。
- **リテラルの リテラルののにはののアンダースコア_**をできます。このをすると、たとえば、リテラルののグループをることができ、コードのがします。
- **インスタンスの**クラスのコンストラクターをびすためになを、パラメーター<>ののセットでえることが出来ます。アングルブラケットのこのペアは、にダイヤモンドとばれています。
- **VarargsメソッドでReifiable Formal Parameters**をするときのコンパイラのとエラーの

バイナリリテラル

```
// An 8-bit 'byte' value:
byte aByte = (byte)0b00100001;

// A 16-bit 'short' value:
short aShort = (short)0b1010000101000101;

// Some 32-bit 'int' values:
int anInt1 = 0b10100001010001011010000101000101;
int anInt2 = 0b101;
int anInt3 = 0B101; // The B can be upper or lower case.

// A 64-bit 'long' value. Note the "L" suffix:
long aLong = 0b1010000101000101101000010100010110100001010001011010000101000101L;
```

try-with-resources ステートメント

ここでは、ファイルからのをみます。 `BufferedReader` インスタンスをしてファイルからデータをみます。 `BufferedReader` は、プログラムのにじるのあるリソースです。

```
static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

ここでは、`try-with-resources` ステートメントでされたリソースは `BufferedReader` です。は、`try` キーワードのにかっこにされます。 `Java SE 7` のクラス `BufferedReader`、`java.lang.AutoCloseable` インタフェースをして `java.lang.AutoCloseable`。 `BufferedReader` インスタンスは `try-with-resource` ステートメントでされているため、`try` ステートメントがにするか、するかにかわらず `IOException` スローする `BufferedReader.readLine` メソッドのとしてじられます。

リテラルの

のは、リテラルでアンダースコアをできるのをしています。

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi = 3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

のにのみアンダースコアをくことができます。のにアンダースコアをすることはできません。

- のまたはに
- リテラルのに
- FまたはLの
- のがされる

インスタンスのための

あなたはうことができます

```
Map<String, List<String>> myMap = new HashMap<>();
```

のわりに

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();
```

ただし、することはできません


```
List<String> list = new ArrayList<>();
list.add("A");

// The following statement should fail since addAll expects
// Collection<? extends String>

list.addAll(new ArrayList<>());
```

それはコンパイルできないためです。ダイヤモンドはメソッドびしでよくくことにしてください。ただし、ダイヤモンドはにのにすることをおめします。

switchの

```
public String getDayOfWeekWithSwitchStatement(String dayOfWeekArg) {
    String typeOfDay;
    switch (dayOfWeekArg) {
        case "Monday":
            typeOfDay = "Start of work week";
            break;
        case "Tuesday":
        case "Wednesday":
        case "Thursday":
            typeOfDay = "Midweek";
            break;
        case "Friday":
            typeOfDay = "End of work week";
            break;
        case "Saturday":
        case "Sunday":
            typeOfDay = "Weekend";
            break;
        default:
            throw new IllegalArgumentException("Invalid day of the week: " + dayOfWeekArg);
    }
    return typeOfDay;
}
```

オンラインでJava SE 7のをむ <https://riptutorial.com/ja/java/topic/8272/java-se-7>の

17: Java SE 8の

き

このトピックでは、Java SE 8のJavaプログラミングにされたのをします。JDBCやJava Virtual Machine JVMなど、のにはくのがありますこのトピックでは

リファレンス [Java SE 8の](#)

Examples

しいJava SE 8プログラミング

- のリリースでは、しいである[Lambda Expressions](#)がされました。それらをする、をメソッドのとしてうことや、コードをデータとしてうことができます。ラムダをすると、メソッドのインタフェースインタフェースとばれますのインスタンスをよりコンパクトにできます。
 - メソッドは、にをつメソッドのみやすいラムダをします。
 - デフォルトのメソッドでは、ライブラリのインタフェースにしいをし、これらのインタフェースのいバージョンにかれたコードとのバイナリをします。
 - Java SE 8のラムダとストリームをしたしくされたAPIでは、ラムダとストリームをするしくされたクラスがされています。
- された - Javaコンパイラは、ターゲットをしてジェネリックメソッドびしのパラメータをします。のターゲット・タイプは、Javaコンパイラがのにじてするデータ・タイプです。たとえば、Java SE 7では、にのターゲットをできます。ただし、Java SE 8では、よりくのコンテキストでにターゲットをできます。
 - ラムダのターゲットタイピング
- アノテーションは、じアノテーションタイプをじまたはタイプにするをします。
- は、だけでなく、がされているでをするをします。プラグブルタイプのシステムでされるこのにより、コードのチェックがされます。
- メソッドのパラメータの - メソッド `java.lang.reflect.Executable.getParameters` して、のメソッドまたはコンストラクタのパラメータのをでき `java.lang.reflect.Executable.getParameters`。クラスのメソッドとコンストラクタは、クラスをし、そのための `Executable.getParameters`。しかし、`.class` ファイルは、デフォルトでは、なパラメータをしなしてください。の `.class` ファイルにパラメータをし、Reflection APIでパラメータをできるようにするには、`javac`コンパイラの `-parameters` オプションをしてソースファイルをコンパイルします。
- Date-time-api - `java.time` しいapiをしました。これをする、タイムゾーンをするはありません。

オンラインでJava SE 8のをむ <https://riptutorial.com/ja/java/topic/8267/java-se-8>

18: JavaStandard Editionのインストール

き

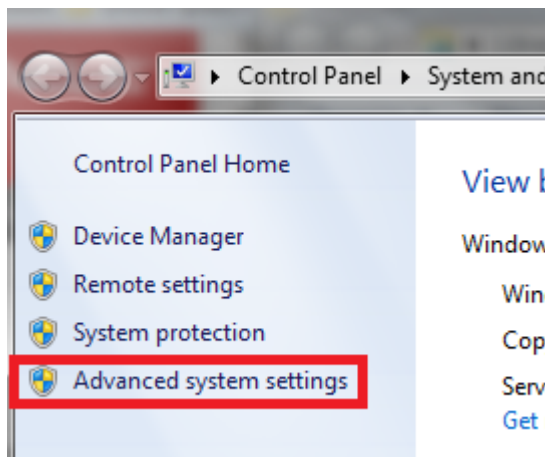
このドキュメントページでは、Windows、Linux、およびmacOSコンピュータにjava standard editionをインストールするにアクセスできます。

Examples

WindowsにインストールしたのPATHとJAVA_HOMEの

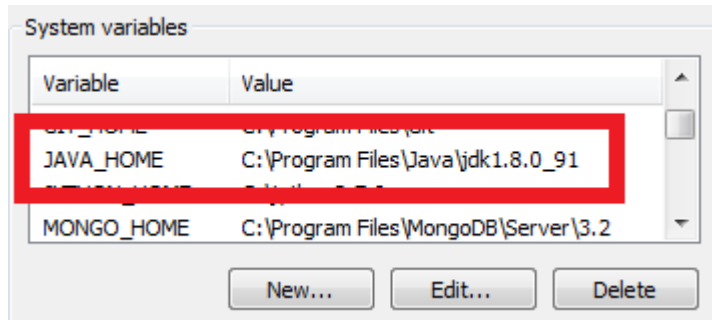
- Oracle JDKがインストールされています。
- JDKはデフォルトのディレクトリにインストールされていました。

1. Windowsエクスプローラをきます。
2. のナビゲーションペインで*This PC* またはWindowsバージョンのは*Computer*をクリックします。のWindowsバージョンでエクスプローラをしないと、よりいがあります Win + Pause
3. しくかれたコントロールパネルウィンドウで、にあるなシステムをクリックします。これにより、システムのプロパティウィンドウがきます。



または、Run Win + R にSystemPropertiesAdvanced をしないと開き、Enterキーを開きます。

4. [システムのプロパティ]の[]タブで、ウィンドウのにある[]ボタンを開きます。
5. がJAVA_HOMEで、がJDKがインストールされたディレクトリへのパスであるシステムの ... ボタンをクリックして、しいシステムをしJAVA_HOME。これらのをしたら、OKを開きます。



6. システムのリストをにスクロールし、 `Path` をします。

7. Windowsは、なプログラムをつけるために `Path` にしています。そのまたはがされた、Windowsがしくしくなるがあります。WindowsがJDKをできるようにするがあります。これをにいて、「Edit ...」ボタンをクリックし、 `Path` をします。
`%JAVA_HOME%\bin;` をし `%JAVA_HOME%\bin;` `Path` のにします。

OracleのソフトウェアがJavaののバージョンを `Path` にするためにされたため、のにするがい - これは、Oracleのにバージョンがされるようにします。

あなたのをチェックする

1. 「スタート」をクリックして `cmd` と `Enter` をして、コマンドプロンプトをきます。
2. プロンプトに `javac -version` します。したは、JDKのバージョンがにされます。

するがあるは、をするにプロンプトをします。これにより、ウィンドウに `Path` しいバージョンがされます。

なJava SE リリースの

1995のJava 1.0リリース、くのJavaのリリースがありました。については、[Javaのバージョン](#)をしてください。しかし、ほとんどのリリースはのEnd of Lifeのをぎました。これは、ベンダーはOracleがリリースのしいをし、バグやセキュリティのパブリック/フリー・パッチをしなくなったことをします。のパッチリリースは、サポートをんでいるやでできますが、ベンダーのにおいわけしてください。

に、するためにされるJava SEリリースは、のパブリックバージョンののアップデートになります。、これはのJava 8リリースをします。Java 9は2017にされるですJava 7はEnd Of Lifeをパスし、20154にされました.Java 7のリリースはおめしません。

このは、すべてのしいJava、およびJavaをぶにされます。サードパーティがするJavaソフトウェアをしたいにもされます。に、よくかれたJavaコードは、Javaのしいリリースでします。ただし、ソフトウェアのリリースノートをし、があるは/サプライヤ/ベンダーにおいわけしてください

いJavaコードベースでしているは、のJavaリリースでコードがされるようにすることをおめしま

す。よりしいJavaリリースのをいつするかをめることは、Javaインストールができない、またはJavaインストールをんでいないをサポートするにするため、よりです。

Javaのリリースとバージョンの

Javaのリリースネーミングはしします。のにすように、けとけの2つのシステムがにあります。

JDKバージョン	マーケティング
jdk-1.0	JDK 1.0
jdk-1.1	JDK 1.1
jdk-1.2	J2SE 1.2
...	...
jdk-1.5	J2SE 1.5、Java SE 5のブランド
JDK-1.6	Java SE 6
jdk-1.7	Java SE 7
jdk-1.8	Java SE 8
jdk-9 ¹	Java SE 9まだリリースされていない

1 - Oracleは、Javaバージョンに「セマンティックバージョン」スキームをするというのをしようとしているようです。らがこれにうかどうかはまだかりません。

マーケティングの「SE」はStandard Editionをします。これは、ほとんどのラップトップ、PC、およびサーバーAndroidをくでJavaをするためのリリースです。

Javaのの2つのがあります。「Java ME」はMicro Edition、「Java EE」はEnterprise Editionです。JavaのAndroidもJava SEときくなります。Java ME、Java EE、およびAndroid Javaは、このトピックのです。

Javaリリースのフルバージョンはのようになります。

```
1.8.0_101-b13
```

これは、JDK 1.8.0、Update 101、Build13をしています。Oracleは、リリース・ノートでこれをするようにしています。

```
Java™ SE Development Kit 8, Update 101 (JDK 8u101)
```

はです。オラクルは、セキュリティパッチ、バグ、によってはをむメジャーリリースのをにして

います。ビルドははがありません。Java 8とJava 1.8 はじことをいます。Java 8は、Java 1.8の「マーケティング」です。

Javaになもの

JDKのインストールとテキストエディタは、Javaののものです。Javaのをうことができるテキストエディタをするのはいいことですが、それをわずにうこともできます。

ただし、なのは、のものもすることをおめします。

- Eclipse、IntelliJ IDEA、NetBeansなどのJava IDE
- Ant、Gradle、MavenなどのJavaビルドツール
- コードベースをするためのバージョンシステムなバックアップとオフサイトレプリケーションを
- テストツールとCIインテグレーションツール

LinuxにJava JDKをインストールする

パッケージマネージャの

OpenJDKまたはOracleのJDKおよびまたはJREリリースは、のLinuxディストリビューションのパッケージマネージャをしてインストールできます。できるはディストリビューションによってなります。

として、ターミナルウィンドウをいてのコマンドをします。 "root"ユーザとしてコマンドをするのになアクセスがあるとしています `sudo` コマンドとじですが、そうでないはシステムのにしてください。

パッケージマネージャをすることをおめします。これはにJavaのインストールをのにつのをにするためです。

`apt-get`、DebianベースのLinuxディストリビューションUbuntuなど

のでOracle Java 8をインストールします。

```
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java8-installer
```

Java 8をにするには、のパッケージをインストールします。

```
$ sudo apt-get install oracle-java8-set-default
```

`.deb` ファイルの

Oracleからダウンロードした`.tar.gz` ファイルから`.deb` ファイルをでするは、のようにします

.tar.gzを./<jdk>.tar.gzダウンロードしたとします。

```
$ sudo apt-get install java-package # might not be available in default repos
$ make-jpkg ./<jdk>.tar.gz          # should not be run as root
$ sudo dpkg -i *j2sdk*.deb
```

これはが ".tar.gz"ファイルとしてされることをとしています。

slackpkg、Slackwareベースの**Linux**ディストリビューション

```
sudo slapt-get install default-jdk
```

yum、RedHat、CentOSなど

```
sudo yum install java-1.8.0-openjdk-devel.x86_64
```

dnf、Fedora

のFedoraリリースでは、yumはdnfってわかりました。

```
sudo dnf install java-1.8.0-openjdk-devel.x86_64
```

のFedoraリリースでは、Java 7のバージョンをインストールするためのパッケージはありません。

pacman、Archベースの**Linux**ディストリビューション

```
sudo pacman -S jdk8-openjdk
```

rootユーザーとしてしているは、sudoをするはありません。

Gentoo Linux

[Gentoo Javaガイド](#)はGentoo Javaチームによってされており、新しいportageパッケージとUSEフラグをむされたwikiページをしています。

RedHat、CentOS、Fedoraへの**Oracle JDK**のインストール

Oracle JDKまたはJREのtar.gzファイルからJDKをインストールする。

1. なりリリースのOracleアーカイブ "tar.gz"ファイルを[Oracle Javaダウンロードサイト](#)からダウンロードします。
2. インストールのディレクトリにします。
3. アーカイブファイルをしします。えば

```
tar xzvf jdk-8u67-linux-x64.tar.gz
```

Oracle Java RPM ファイルからのインストール

1. なりリリースになRPMファイルを[Oracle Javaダウンロードサイト](#)からします。
2. rpmコマンドをしてインストールします。えは

```
$ sudo rpm -ivh jdk-8u67-linux-x644.rpm
```

WindowsでのJava JDKまたはJREのインストール

WindowsプラットフォームではOracle JDKおよびJREのみできます。インストールはです。

1. Oracle Java [ダウンロードページ](#)にアクセスしてください
2. JDKボタン、JREボタン、またはServer JREボタンのいずれかをクリックします。Javaをしてするには、JDKがです。JDKとJREのいをするには、[ここ](#)をしてください
3. ダウンロードするバージョンまでスクロールします。にえは、のものをおめします
4. [Accept License Agreement]ラジオボタンをします。
5. Windows x8632ビットまたはWindows x6464ビットインストーラをダウンロードします。
6. あなたのバージョンのWindowsでは、のでインストーラをしてください。

コマンドプロンプトをしてWindowsにJavaをインストールするのは、Chocolateyをすることです。

1. <https://chocolatey.org/>からChocolateyをインストールしてください
2. cmdインスタンスをきます。たとえば、Win + Rキーをしてから、「」ウィンドウに「cmd」としてEnterキーをします。
3. cmdインスタンスで、のコマンドをしてJava 8 JDKをダウンロードしてインストールします。

```
C:\> choco install jdk8
```

ポータブルのと

VMなどのられたをつシステムにJDK / JREをインストールするや、JDK / JREののバージョンまたはアーキテクチャx64 / x86をインストールしてしたいがあります。は、インストーラ.EXEをダウンロードするまでわりません。そののはのとおりでこのはJDK / JRE 7にされますが、いバージョンではフォルダとファイルのがしなります。

1. Javaバイナリがにするなにファイルをします。
- 2.

られたをっているは、7-Zipまたはそのポータブルをインストールしてください。

3. 7-Zipをして、ファイルをJavaインストーラEXEからそのにします。

4. エクスプローラのフォルダにShiftとRight-Clickしながらコマンドプロンプトを開くか、どこからでもそのにします。

5. しくしたフォルダにします。フォルダがjdk-7u25-windows-x64です。 cd jdk-7u25-windows-x64
します。に、のコマンドをにします。

```
cd .rsrc\JAVA_CAB10
```

```
extrac32 111
```

6. これにより、そのにtools.zipファイルがされます。 tools.zipを7-Zipでし、そののファイルがディレクトリのtoolsにされるようにしtools。

7. これらのコマンドは、すでにいているコマンドプロンプトでします。

```
cd tools
```

```
for /r %x in (*.pack) do .\bin\unpack200 -r "%x" "%~dx%~px%~nx.jar"
```

8. コマンドがするのをちます。 toolsのを、バイナリをするにコピーします。

これで、にインストールするのあるJDK / JREののバージョンをインストールできます。

の <http://stackoverflow.com/a/6571736/1448252>

MacOSにJava JDKをインストールする

Oracle Java 7およびJava 8

MacOSのJava 7およびJava 8は、Oracleからできます。このOracleのページは、Java for Macにするくのにえています。セキュリティのから、7u25よりのJava 7はAppleによってにされていることにしてください。

に、Oracle Javaバージョン7では、MacOS 10.7.3をするIntelベースのMacがです。

Oracle Javaのインストール

MacOSのJava 7および8 JDKおよびJREインストーラは、OracleのWebサイトからダウンロードできます。

- Java 8 - [Java SEのダウンロード](#)
- Java 7 - [Oracle Javaアーカイブ](#)。

するパッケージをダウンロードした、パッケージをダブルクリックし、のインストールプロセスをします。JDKはここにインストールするがあります

```
/Library/Java/JavaVirtualMachines/<version>.jdk/Contents/Home
```

インストールされているバージョンにします。

コマンドラインのりえ

Javaがインストールされると、インストールされたバージョンがデフォルトとしてされます。
なったものにりえるには

```
export JAVA_HOME=/usr/libexec/java_home -v 1.6 #Or 1.7 or 1.8
```

いやすさのため、のを ~/.bash_profile することができますデフォルトのBashシェルをする。

```
function java_version {
    echo 'java -version';
}

function java_set {
    if [[ $1 == "6" ]]
    then
        export JAVA_HOME='/usr/libexec/java_home -v 1.6';
        echo "Setting Java to version 6..."
        echo "$JAVA_HOME"
    elif [[ $1 == "7" ]]
    then
        export JAVA_HOME='/usr/libexec/java_home -v 1.7';
        echo "Setting Java to version 7..."
        echo "$JAVA_HOME"
    elif [[ $1 == "8" ]]
    then
        export JAVA_HOME='/usr/libexec/java_home -v 1.8';
        echo "Setting Java to version 8..."
        echo "$JAVA_HOME"
    fi
}
```

MacOSのApple Java 6

いバージョンのmacOS10.11 El Capitanでは、AppleのJava 6のリリースがあらかじめインストールされています。インストールされているは、のにあります。

```
/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
```

Java 6はなくなってしまったので、しいバージョンにアップグレードすることをおめします。
Apple WebサイトにApple Java 6をインストールするについては、さらにしいがあります。

LinuxでのJavaバージョンのとりえ

の

くのLinuxディストリビューションでは、なるバージョンのコマンドをりえるために`alternatives`コマンドをしています。これをして、マシンにインストールされているなるバージョンのJavaをりえることができます。

1. コマンドシェルでは、\$ JDKをしくインストールされたJDKのパスにします。えば

```
$ JDK=/Data/jdk1.8.0_67
```

2. `alternatives --install`をして、Java SDKのコマンドを`alternatives --install`にします。

```
$ sudo alternatives --install /usr/bin/java java $JDK/bin/java 2
$ sudo alternatives --install /usr/bin/javac javac $JDK/bin/javac 2
$ sudo alternatives --install /usr/bin/jar jar $JDK/bin/jar 2
```

々。

のようにして、Javaコマンドのさまざまなバージョンをりえることができます。

```
$ sudo alternatives --config javac

There is 1 program that provides 'javac'.

   Selection      Command
-----
*+ 1              /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.101-1.b14.fc23.x86_64/bin/javac
   2              /Data/jdk1.8.0_67/bin/javac

Enter to keep the current selection[+], or type selection number: 2
$
```

`alternatives`のについては、[8](#)のマニュアルをしてください。

アーチベースのインストール

Arch Linuxベースのインストールでは、`archlinux-java`コマンドを`archlinux-java` Javaバージョンをりえることができます。

インストールされたの

```
$ archlinux-java status
Available Java environments:
  java-7-openjdk (default)
  java-8-openjdk/jre
```

のをりえる

```
# archlinux-java set <JAVA_ENV_NAME>
```

えば

```
# archlinux-java set java-8-openjdk/jre
```

は、 [Arch Linux Wiki](#)をしてください。

Linuxでのインストールのチェックと

Java SDKをインストールしたら、がっていることをすることをおめします。これは、のユーザーアカウントをして、の2つのコマンドをすることです。

```
$ java -version
$ javac -version
```

これらのコマンドは、シェルのコマンドパスにあるJREとJDKのバージョンをします。JDK / JREのバージョンをします。

- のコマンドのいずれかがし、「コマンドが見つかりません」とされた、JREまたはJDKはパスにまったくしません。の「**PATH**の」をしてください。
- のいずれかのコマンドでしていたものとなるバージョンのがされた、パスまたは ""システムのいずれかをすることがあります。のチェックにく
- しいバージョンがされていれば、ほぼです。**JAVA_HOME**のチェックにスキップする

PATHの

でパスにjavaまたはjavacがない、なは、パスにすることです。

まず、Javaをインストールしたをつけます。「**Java**はどこにインストールされましたかがあるはをください。

に、bashがコマンドシェルであるとして、 ~/.bash_profileまたは ~/.bashrc シェルとしてBashをするのにテキストエディタをしてのをします。

```
JAVA_HOME=<installation directory>
PATH=$JAVA_HOME/bin:$PATH

export JAVA_HOME
export PATH
```

... <installation directory>をJavaインストールディレクトリのパスにきえます。では、インストールディレクトリにbinディレクトリがあり、binディレクトリに、しようとしているjavaコマンドとjavacコマンドがまれていることをとしています。

に、したばかりのファイルをソースし、のシェルのをします。

```
$ source ~/.bash_profile
```

に、`java`と`javac`バージョンチェックをります。それでもがあるは、`which java`と`which javac`をし`which java`、がしくされていることをします。

に、ログアウトしてログインし、されたがすべてのシェルに`ptopagate`するようにします。あなたはわれるべきです。

の

`java -version`または`javac -version`していても、しないバージョンをしたは、コマンドのをするがあります。 `which`と`ls -l`をとってのようにべます

```
$ ls -l `which java`
```

がのようになっているは、

```
lrwxrwxrwx. 1 root root 22 Jul 30 22:18 /usr/bin/java -> /etc/alternatives/java
```

`alternatives`バージョンのりえがされています。 `PATH`するかどうかをするがあります。

- [LinuxでのJavaバージョンのとりえ](#)
- の「`PATH`の」をしてください。

Javaはどこにインストールされましたか

Javaは、インストールにじて、さまざまにインストールできます。

- Oracle RPMはJavaインストールを `"/usr/java"`にきます。
- Fedoraでは、デフォルトのは `"/usr/lib/jvm"`です。
- JavaがZIPまたはJARファイルからでインストールされた、インストールはどこにでもできます。

インストールディレクトリをつけにくいは、 `find` または `slocate` をしてコマンドをつけることをおめします。えば

```
$ find / -name java -type f 2> /dev/null
```

これにより、システムの`java`とばれるすべてのファイルのパスがされます。 エラーを `"/dev/null"`にリダイレクトすると、アクセスできないファイルやディレクトリにするメッセージはされなくなります。

の**tar**ファイルをして**Linux**に**oracle java**をインストールする

のにつて、のtarファイルからOracle JDKをインストールします。

1. のtarファイルを[ここ](#)からダウンロードして[ください](#) - はJava SE Development Kit 8u112で

す。

2. あなたはsudoのがです

```
sudo su
```

3. jdk installのためのdirをします

```
mkdir /opt/jdk
```

4. それにダウンロードされたtarをする

```
tar -zxvf jdk-8u5-linux-x64.tar.gz -C /opt/jdk
```

5. ファイルがされているかどうかをします。

```
ls /opt/jdk
```

6. Oracle JDKをデフォルトのJVMとしてする

```
update-alternatives --install /usr/bin/java java /opt/jdk/jdk1.8.0_05/bin/java 100
```

そして

```
update-alternatives --install /usr/bin/javac javac /opt/jdk/jdk1.8.0_05/bin/javac 100
```

7. Javaバージョンをする

```
java -version
```

される

```
java version "1.8.0_111"  
Java(TM) SE Runtime Environment (build 1.8.0_111-b14)  
Java HotSpot(TM) 64-Bit Server VM (build 25.111-b14, mixed mode)
```

オンラインでJavaStandard Editionのインストールをむ

<https://riptutorial.com/ja/java/topic/4754/java-standard-edition-のインストール>

19: java.util.Objects クラス

Examples

オブジェクトの**NULL**チェックのな

ヌルチェックインメソッドの

```
Object nullableObject = methodReturnObject();
if (Objects.isNull(nullableObject)) {
    return;
}
```

not null チェックインメソッドの

```
Object nullableObject = methodReturnObject();
if (Objects.nonNull(nullableObject)) {
    return;
}
```

ストリームAPIでの**Objects.nonNull**メソッドの

コレクションのヌルチェックのための

```
List<Object> someObjects = methodGetList();
for (Object obj : someObjects) {
    if (obj == null) {
        continue;
    }
    doSomething(obj);
}
```

Objects.nonNullメソッドとJava8 Stream APIをすると、のようにすることができます。

```
List<Object> someObjects = methodGetList();
someObjects.stream()
    .filter(Objects::nonNull)
    .forEach(this::doSomething);
```

オンラインでjava.util.Objectsクラスをむ <https://riptutorial.com/ja/java/topic/5768/java-util-objects>
クラス

20: JavaBean

き

JavaBeans™は、インスタンスBeanをさまざまなコンテキストでし、JavaコードをにせずにさまざまなツールをできるJavaクラスAPIをするためのパターンです。パターンは、プロパティのゲッターとセッターをするためのコンベンション、コンストラクターの、およびイベントリスナーAPIのためのからされています。

- **JavaBean** プロパティの
 - プロパティがブールでない、getterメソッドのはgetでなければなりません。たとえば、getSizeは "size" というのプロパティのなJavaBeansゲッタです。 sizeというのはないことにしてください。プロパティのはgetterとsetterからされ、クラスのはされません。 getSizeからすものはあなたです。
 - プロパティがブールの、getterメソッドのはgetまたはisのいずれかです。たとえば、getStoppedまたはisStoppedはともbooleanプロパティのなJavaBeansです。
 - setterメソッドのプレフィックスをするがあります。たとえば、setSizeはsizeというプロパティのなJavaBeanです。
 - getterメソッドまたはsetterメソッドのをするには、プロパティのをにし、なget、is、またはsetにします。
 - Setterメソッドのシグネチャは、voidのりのとそのプロパティのをすでpublicとマークするがあります。
 - Getterメソッドのシグネチャは、publicとマークし、をらず、そのプロパティのsetterメソッドののとするりのをたなければなりません。
- **JavaBean** リスナーの
 - リスナーをイベントソースに「」するためにされるリスナーメソッドは、addとそれにくリスナータイプをするがあります。たとえば、addActionListenerは、イベントソースがのアクションイベントをできるようにするメソッドのなです。
 - リスナーを「」するためにされるリスナーメソッドは、remove、そのにリスナータイプをするがありますaddメソッドとじルールを。
 - またはされるリスナーのは、メソッドのとしてされるがあります。
 - リスナーのメソッドは、 "Listener" というでわらなければなりません。

クラスがJava Beanになるためには、このにわなければなりません -

- すべてのプロパティはプライベートで、getterおよびsetterをしてのみアクセスでなければなりません。
- パブリックにのいないコンストラクタがです。
- `java.io.Serializable` インタフェースをするがあります。

Examples

なJava Bean

```
public class BasicJavaBean implements java.io.Serializable{

    private int value1;
    private String value2;
    private boolean value3;

    public BasicJavaBean(){}

    public void setValue1(int value1){
        this.value1 = value1;
    }

    public int getValue1(){
        return value1;
    }

    public void setValue2(String value2){
        this.value2 = value2;
    }

    public String getValue2(){
        return value2;
    }

    public void setValue3(boolean value3){
        this.value3 = value3;
    }

    public boolean isValue3(){
        return value3;
    }

}
```

オンラインでJavaBeanをむ <https://riptutorial.com/ja/java/topic/8157/javabean>

21: Javaエージェント

Examples

エージェントでクラスをする

まず、されているエージェントがManifest.mfにのをつことをします。

```
Can-Redefine-Classes: true
Can-Transform-Classes: true
```

Javaエージェントをすると、エージェントはInstrumentationクラスにアクセスします。Instrumentationでは、*addTransformerClassFileTransformer*トランスフォーマをびすことができます。ClassFileTransformersをすると、クラスのバイトをきえることができます。クラスには、クラスをロードするClassLoader、クラスの、そのjava.lang.Classインスタンス、ProtectionDomain、にクラスのバイトをするメソッドが1つしかありません。

これはのようになります。

```
byte[] transform(ClassLoader loader, String className, Class<?> classBeingRedefined,
    ProtectionDomain protectionDomain, byte[] classfileBuffer)
```

にバイトからクラスをするにはが掛かることがあります。これをするために、クラスバイトをよりしやすいものにするためにできるライブラリがあります。

このではASMをしますが、JavassistやBCELのようなのものがあります。

```
ClassNode getNode(byte[] bytes) {
    // Create a ClassReader that will parse the byte array into a ClassNode
    ClassReader cr = new ClassReader(bytes);
    ClassNode cn = new ClassNode();
    try {
        // This populates the ClassNode
        cr.accept(cn, ClassReader.EXPAND_FRAMES);
        cr = null;
    } catch (Exception e) {
        e.printStackTrace();
    }
    return cn;
}
```

ここから、ClassNodeオブジェクトをすることができます。これにより、フィールド/メソッドのアクセスをにできます。また、メソッドのバイトコードをするASMのTree APIにもアクセスできます。

がしたら、のメソッドでClassNodeをバイトにしてから、メソッドですことができます。

```

public static byte[] getNodeBytes(ClassNode cn, boolean useMaxs) {
    ClassWriter cw = new ClassWriter(useMaxs ? ClassWriter.COMPUTE_MAXS :
ClassWriter.COMPUTE_FRAMES);
    cn.accept(cw);
    byte[] b = cw.toByteArray();
    return b;
}

```

にエージェントをする

にエージェントをJVMにすることができます。エージェントをみむには、Attach APIの *VirtualMachine.attachString id*をするがあります。コンパイルされたエージェントjarをロードするには、のメソッドをします。

```

public static void loadAgent(String agentPath) {
    String vmName = ManagementFactory.getRuntimeMXBean().getName();
    int index = vmName.indexOf('@');
    String pid = vmName.substring(0, index);
    try {
        File agentFile = new File(agentPath);
        VirtualMachine vm = VirtualMachine.attach(pid);
        vm.loadAgent(agentFile.getAbsolutePath(), "");
        VirtualMachine.attach(vm.id());
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

```

これは、ロードされたエージェントの *premainString agentArgs*、Instrumentation *inst*をびすのではなく、わりに *agentmainString agentArgs*、Instrumentation *inst*をびします。これはエージェントクラス *Manifest.mf*にエージェントクラスをするがあります。

エージェントの

*Premain*クラスには、メソッド "*premainString agentArgs Instrumentation inst*"がまれます。

にをします。

```

import java.lang.instrument.Instrumentation;

public class PremainExample {
    public static void premain(String agentArgs, Instrumentation inst) {
        System.out.println(agentArgs);
    }
}

```

jarファイルにコンパイルされたとき、*Manifest*をき、*Premain-Class*をつことをします。

にをします。

```
Premain-Class: PremainExample
```

のJavaプログラム "myProgram"でエージェントをするには、JVMにエージェントをするがあります。

```
java -javaagent:PremainAgent.jar -jar myProgram.jar
```

オンラインでJavaエージェントをむ <https://riptutorial.com/ja/java/topic/1265/javaエージェント>

22: Javaコードの

き

Javaコードのドキュメントは、[javadoc](#)をしてされることがよくあります。Javadocは、JavaソースコードからHTMLの[APIドキュメント](#)をするで、Sun Microsystemsによってされました。HTMLをすると、するをにハイパーリンクすることができるというがられます。

- `/**` - クラス、フィールド、メソッド、またはパッケージのJavaDocの
- `@author` //クラス、インタフェースまたはenumのにをける。それはです。
- `@version` //そのクラス、インタフェース、またはenumのバージョン。それはです。ソースソフトウェアがチェックアウトにするために、IやGのようなマクロをすることができます。
- `@param` //メソッドまたはコンストラクタのパラメータをする。パラメータごとに`@param`タグを1つします。
- `@return` // voidのメソッドのりのをします。
- `@exception` //メソッドやコンストラクタからスローされるをします。されなければならぬいは、ここにリストされていなければなりません。にじて、
ArrayIndexOutOfBoundsExceptionのように、キャッチするのなないものもめることができます。スローできるごとに1つの`@exception`をします。
- `@throws` // `@exception`とじです。
- `@see` //メソッド、フィールド、クラス、またはパッケージにリンクします。
package.Classかのでしてください。
- `@since` //このメソッド、フィールド、またはクラスがされたとき。たとえば、
[java.util.Optional <T>](#)のようなクラスのJDK-8です。
- `@serial`、`@serialField`、`@serialData` // `serialVersionUID`をするためにします。
- `@deprecated` //クラス、メソッド、またはフィールドをとしてマークする。たとえば、
[java.io.StringBufferInputStream](#)があります。のクラスのなリストは[こちらをご覧ください](#)。
- `{@link}` // `@see`とていますが、カスタムテキストでうことができます`{@link #setDefaultCloseOperationint closeOperation}`については、JFrame
`setDefaultCloseOperation`をしてください。
- `{@linkplain}` //コードフォントがなく、`{@link}`とています。
- `{@code}` // HTMLタグなどのリテラルコード。`{@code <html> </html>}`。しかし、これはモノスペースフォントをします。モノスペースフォントなしでじをるには、`{@literal}`をしてください。
- `{@literal}` //モノスペースのフォントをわずに`{@code}`とじです。
- `{@value}` //フィールドのをします。JFrame.EXIT_ON_CLOSEのは`{@value}`です。または、
のフィールドにリンクすることができますアプリ`{@value AppConstantsAPP_NAME}`をします。
- `{@docRoot}` //のファイルをしたJavaDoc HTMLのルートフォルダ。`<a href="{@docRoot}/credits.html">クレジット</a>`
- HTMLはされています`<code> "Hi cookies" .substring3</code>`。

- `*/` - Javadocのわり

Javadocは、コードのコメントをHTMLドキュメントにできるJDKにまれているツールです。[Java API](#)は、Javadocをしてされました。サードパーティのライブラリのくのドキュメントでもじことがえます。

Examples

クラス

すべてのJavadocコメントは、ブロックコメントとアスタリスク `/**` でまり、ブロックコメントが `*/` のときにします。で、はのとアスタリスクでめることができます。ドキュメンテーション・ファイルがされると、これらはされます。

```
/**
 * Brief summary of this class, ending with a period.
 *
 * It is common to leave a blank line between the summary and further details.
 * The summary (everything before the first period) is used in the class or package
 * overview section.
 *
 * The following inline tags can be used (not an exhaustive list):
 * {@link some.other.class.Documentation} for linking to other docs or symbols
 * {@link some.other.class.Documentation Some Display Name} the link's appearance can be
 * customized by adding a display name after the doc or symbol locator
 * {@code code goes here} for formatting as code
 * {@literal <>[]()foo} for interpreting literal text without converting to HTML markup
 * or other tags.
 *
 * Optionally, the following tags may be used at the end of class documentation
 * (not an exhaustive list):
 *
 * @author John Doe
 * @version 1.0
 * @since 5/10/15
 * @see some.other.class.Documentation
 * @deprecated This class has been replaced by some.other.package.BetterFileReader
 *
 * You can also have custom tags for displaying additional information.
 * Using the @custom.<NAME> tag and the -tag custom.<NAME>:htmltag:"context"
 * command line option, you can create a custom tag.
 *
 * Example custom tag and generation:
 * @custom.updated 2.0
 * Javadoc flag: -tag custom.updated:a:"Updated in version:"
 * The above flag will display the value of @custom.updated under "Updated in version:"
 */
public class FileReader {
}
```

EnumsとInterfacesにも、Classesされているのとじタグとをできます。

メソッドドキュメント

すべてのJavadocコメントは、ブロックコメントとアスタリスク `/**` でまり、ブロックコメントが `*/` のときにします。で、はのとアスタリスクでめることができます。ドキュメンテーション・ファイルがされると、これらはされます。

```
/**
 * Brief summary of method, ending with a period.
 *
 * Further description of method and what it does, including as much detail as is
 * appropriate. Inline tags such as
 * {@code code here}, {@link some.other.Docs}, and {@literal text here} can be used.
 *
 * If a method overrides a superclass method, {@inheritDoc} can be used to copy the
 * documentation
 * from the superclass method
 *
 * @param stream Describe this parameter. Include as much detail as is appropriate
 *             Parameter docs are commonly aligned as here, but this is optional.
 *             As with other docs, the documentation before the first period is
 *             used as a summary.
 *
 * @return Describe the return values. Include as much detail as is appropriate
 *         Return type docs are commonly aligned as here, but this is optional.
 *         As with other docs, the documentation before the first period is used as a
 *         summary.
 *
 * @throws IOException Describe when and why this exception can be thrown.
 *             Exception docs are commonly aligned as here, but this is
 *             optional.
 *             As with other docs, the documentation before the first period
 *             is used as a summary.
 *             Instead of @throws, @exception can also be used.
 *
 * @since 2.1.0
 * @see some.other.class.Documentation
 * @deprecated Describe why this method is outdated. A replacement can also be specified.
 */
public String[] read(InputStream stream) throws IOException {
    return null;
}
```

フィールド

すべてのJavadocコメントは、ブロックコメントとアスタリスク `/**` でまり、ブロックコメントが `*/` のときにします。で、はのとアスタリスクでめることができます。ドキュメンテーション・ファイルがされると、これらはされます。

```
/**
 * Fields can be documented as well.
 *
 * As with other javadocs, the documentation before the first period is used as a
 * summary, and is usually separated from the rest of the documentation by a blank
 * line.
 *
 * Documentation for fields can use inline tags, such as:
 * {@code code here}
 * {@literal text here}
```

```

* {@link other.docs.Here}
*
* Field documentation can also make use of the following tags:
*
* @since 2.1.0
* @see some.other.class.Documentation
* @deprecated Describe why this field is outdated
*/
public static final String CONSTANT_STRING = "foo";

```

パッケージドキュメント

Java SE 5

とばれるファイルのJavadocにパッケージレベルのドキュメントをすることがである`package-info.java`。このファイルはのようにフォーマットするがあります。するとアスタリスク。、のによりにされます。

```

/**
 * Package documentation goes here; any documentation before the first period will
 * be used as a summary.
 *
 * It is common practice to leave a blank line between the summary and the rest
 * of the documentation; use this space to describe the package in as much detail
 * as is appropriate.
 *
 * Inline tags such as {@code code here}, {@link reference.to.other.Documentation},
 * and {@literal text here} can be used in this documentation.
 */
package com.example.foo;

// The rest of the file must be empty.

```

の、このファイル`package-info.java`をJavaパッケージ`com.example.foo`フォルダのにくがあります。

リンク

のJavadocへのリンクは`@link`タグでい`@link`

```

/**
 * You can link to the javadoc of an already imported class using {@link ClassName}.
 *
 * You can also use the fully-qualified name, if the class is not already imported:
 * {@link some.other.ClassName}
 *
 * You can link to members (fields or methods) of a class like so:
 * {@link ClassName#someMethod()}
 * {@link ClassName#someMethodWithParameters(int, String)}
 * {@link ClassName#someField}
 * {@link #someMethodInThisClass()} - used to link to members in the current class
 *
 * You can add a label to a linked javadoc like so:
 * {@link ClassName#someMethod() link text}
 */

```


You can link to the javadoc of an already imported class using [ClassName](#).

You can also use the fully-qualified name, if the class is not already imported: [some.other.ClassName](#)

You can link to members (fields or methods) of a class like so:

[ClassName.someMethod\(\)](#)

[ClassName.someMethodWithParameters\(int, String\)](#)

[ClassName.someField](#)

[someMethodInThisClass\(\)](#) - used to link to members in the current class

You can add a label to a linked javadoc like so: [link text](#)

`@see` タグでは、`See also` セクションにをできます。 `@param` や `@return` とに、されるはありません。では、 `@return` にするがあります。

```
/**
 * This method has a nice explanation but you might found further
 * information at the bottom.
 *
 * @see ClassName#someMethod()
 */
```

This method has a nice explanation but you might found further

See Also:

[ClassName.someMethod\(\)](#)

リソースへのリンクをするは、HTMLの `<a>` タグをします。どこでもインラインでうことも、`@link @see` タグと `@see` タグのでもできます。

```
/**
 * Wondering how this works? You might want
 * to check this <a href="http://stackoverflow.com/">great service</a>.
 *
 * @see <a href="http://stackoverflow.com/">Stack Overflow</a>
 */
```

Wondering how this works? You might want to check this [great service](#).

See Also:

[Stack Overflow](#)

コマンドラインからのJavadocsのビルド

くのIDEはJavadocからにHTMLをするためのサポートをします。いくつかのビルドツール [Maven](#) や [Gradle](#) などには、HTMLをできるプラグインもあります。

ただし、これらのツールはJavadoc HTMLをするはありません。これは、コマンドライン `javadoc` ツールをしてうことができます。

このツールのもないはのとおりです。

```
javadoc JavaFile.java
```

JavaFile.javaのJavadocコメントからHTMLをします。

[source-directory]にあるすべてのJavaファイルをにみむコマンドラインツールをにすると、[package.name]とすべてのサブパッケージのドキュメントがされ、されたHTMLが[docs-directory]のとおりです。

```
javadoc -d [docs-directory] -subpackages -sourcepath [source-directory] [package.name]
```

インラインコードのドキュメント

Javadocのドキュメントコードにも、コードはインラインでできます。

のコメントは、//によってされ、じにあるのにかけますが、にはかけません。

```
public void method() {  
  
    //single line comment  
    someMethodCall(); //single line comment after statement  
  
}
```

コメントは/*と*/にされます。それらはのにまたがることができ、のにかれることさえあります。

```
public void method(Object object) {  
  
    /*  
     multi  
     line  
     comment  
    */  
    object/*inner-line-comment*/.method();  
}
```

JavaDocsは、/**でまるなのコメントです。

あまりにもくのインラインコメントはコードのをさせるがあるため、コードがにではないや、のがらかでないはまばらにするがあります。

1コメントのとしては、TAGをすることがあります。くてコンベンションドリブンなキーワードです。いくつかのでは、そのようなコメントのためののをしています。なは

- //TODO
- //FIXME

または、Jiraのためにをする

- //PRJ-1234

ドキュメントのコードスニペット

ドキュメントのコードをくは、 `{@code }` です。 `<pre></pre>`にのコードラップがある。

```
/**
 * The Class TestUtils.
 * <p>
 * This is an {@code inline("code example")}.
 * <p>
 * You should wrap it in pre tags when writing multiline code.
 * <pre>{@code
 *   Example example1 = new FirstLineExample();
 *   example1.butYouCanHaveMoreThanOneLine();
 * }</pre>
 * <p>
 * Thanks for reading.
 */
class TestUtils {
```

によっては、javadocコメントのになコードをれるがあるかもしれません。 `@`はにがあります。い
`<code>`タグを `{@literal }` とにすることで、このはされます。

```
/**
 * Usage:
 * <pre><code>
 * class SomethingTest {
 *   {@literal @}Rule
 *   public SingleTestRule singleTestRule = new SingleTestRule("test1");
 *
 *   {@literal @}Test
 *   public void test1() {
 *       // only this test will be executed
 *   }
 *
 *   ...
 * }
 * </code></pre>
 */
class SingleTestRule implements TestRule { }
```

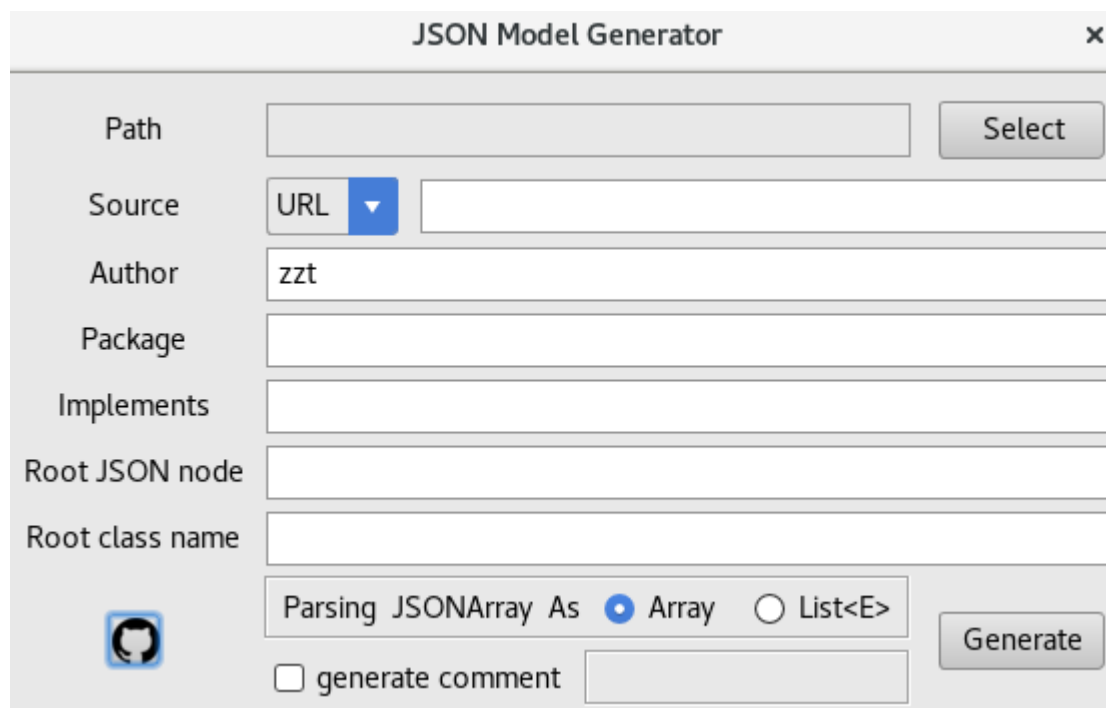
オンラインでJavaコードのをむ <https://riptutorial.com/ja/java/topic/140/javaコードの>

23: Javaコードの

Examples

JSONからPOJOをする

- IntelliJのIntelliJのJSON Model Genratorプラグインをインストールします。
- プラグインを 'ツール' からする
- UIのフィールドをのようになります 'Path'、'Source'、'Package'がです。



The screenshot shows the 'JSON Model Generator' dialog box in IntelliJ. It has a title bar with a close button (X). The dialog contains several input fields and buttons:

- Path:** A text input field with a 'Select' button to its right.
- Source:** A dropdown menu currently showing 'URL' and an empty text input field to its right.
- Author:** A text input field containing the text 'zzt'.
- Package:** An empty text input field.
- Implements:** An empty text input field.
- Root JSON node:** An empty text input field.
- Root class name:** An empty text input field.
- Parsing JSONArray As:** Two radio buttons, 'Array' (which is selected) and 'List<E>'.
- generate comment:** A checkbox that is currently unchecked.
- Buttons:** A 'Generate' button at the bottom right and a small GitHub icon at the bottom left.

- [Generate]ボタンをクリックするとです。

オンラインでJavaコードのをむ <https://riptutorial.com/ja/java/topic/9400/javaコードの>

24: Java コマンド - 'java' と 'javaw'

- `java [ <opt> ... ] <class-name> [ <argument> ... ]`
- `java [ <opt> ... ] -jar <jar-file-pathname> [ <argument> ... ]`

`java` コマンドは、コマンドラインからJavaアプリケーションをするためにされます。Java SE JREまたはJDKのとしてできます。

Windowsシステムでは、`java` コマンドに2つのバリエーションがあります。

- `java` は、新しいコンソールウィンドウでアプリケーションをします。
- `javaw` バリエーションは、新しいコンソールウィンドウをせずにアプリケーションをします。

のシステムLinux、Mac OSX、UNIXなどでは、`java` コマンドだけがされ、新しいコンソールウィンドウはしません。

の<opt>は、`java` コマンドのオプションをします。もにされるオプションには、「Javaオプション」と「ヒープとスタックサイズのオプション」のトピックがあります。そののについては、[JVM Flags](#) トピックをしてください。

Examples

なJARファイルの

なJARファイルは、JavaコードをなファイルにアSEMBLするもなです。*JARファイルのはのトピックでカバーするがあります*

パス<jar-path>をつなJARファイルがあるとすると、のようにできます。

```
java -jar <jar-path>
```

コマンドにコマンドラインがなは、<jar-path>にします。えは

```
java -jar <jar-path> arg1 arg2 arg3
```

`java` コマンドでのJVMオプションをするがあるは、`-jar` オプションのにするがあります。`-jar` をする、`-cp` / `-classpath` オプションはされることにしてください。アプリケーションのクラスパスは、JARファイルマニフェストによってされます。

「メイン」クラスをしたJavaアプリケーションの

アプリケーションがなJARとしてパッケージされていないは、`java` コマンドラインで[エントリーポイントクラス](#)のをするがあります。

HelloWorld クラスの

「HelloWorld」のについては、「[しいJavaプログラムの](#)」をしてください。これは、エントリーポイントのをたす HelloWorld というのクラスでされています。

コンパイルされた "HelloWorld.class" ファイルがカレントディレクトリにあるとすると、のように入ることができます

```
java HelloWorld
```

すべきなのはのとおりです。

- ".class" ファイルまたは ".java" ファイルのパスではなく、クラスのをするがあります。
- クラスがパッケージでされているほとんどのJavaクラスがそうであるように、 java コマンドにするクラスはなクラスでなければなりません。たとえば、 SomeClass が com.example パッケージでされている、なクラスは com.example.SomeClass ます。

クラスパスの

java -jar コマンドでしているを java -jar 、 java コマンドはクラスパスをしてロードするクラスをします。 [クラスパス](#) をしてください。 のコマンドは、 のディレクトリをむまたはむデフォルトのクラスパスにしています。 -cp オプションをしてするクラスパスをすることで、これをよりに入ることができます。

```
java -cp . HelloWorld
```

これは、 のディレクトリ "。" がしているものをクラスパスののエントリーにすることをします。

-cp は、 java コマンドによってされるオプションです。 java コマンドをとするすべてのオプションは、クラスのにあるがあります。クラスのもの、Javaアプリケーションのコマンドラインとしてわれ、 main メソッドにされる String[] でアプリケーションにされます。

-cp オプションをしないと、 java は CLASSPATH でされたクラスパスをします。そのがされていないかであれば、 java はデフォルトのクラスパスとして "。" をします。

エントリーポイントクラス

Javaのエントリーポイントクラスには、のシグネチャとをつ main メソッドがあります。

```
public static void main(String[] args)
```

Sidenoteのみのせいで、 (String args[])

java コマンドがマシンをすると、されたエントリーポイントクラスがロードされ、 main をしようと

します。すると、コマンドラインからのがJava Stringオブジェクトにされ、にアセンブルされます。はmain、のようにびされ、はなりません nullとどんなまれませんnullエントリを。

なエントリポイントクラスのメソッドでは、のをうがあります。

- main とを
- publicとstatic
- voidをす
- String[]をつのをちます。はしなければならず、はすることはできません。
- ジェネリックパラメータはされません。
- ジェネリックではない、トップレベルネストされていない、またはではないのみクラス

クラスをpublicとしてするのはpublicが、これはにはではありません。Java 5では、mainメソッドのはではなくString varargsであるがあります。mainはオプションでをスローすることができ、そのパラメータのはでもかまいませんが、はargsです。

JavaFX エントリーポイント

Java 8では、javaコマンドをしてJavaFXアプリケーションをすることもできます。JavaFXのは、にされてのJavaFXタグが、JavaFXのエントリー・ポイントは、をうがあります。

- javafx.application.Applicationする
- publicしabstractではない
- ジェネリックでもネストでもない
- またはなpublicなしのコンストラクタをつ

'java' コマンドのトラブルシューティング

ここでは、'java'コマンドをするのなエラーについてします。

"コマンドが見つかりません"

のようなエラーメッセージがされた

```
java: command not found
```

javaコマンドをしようとする、シェルのコマンドパスにjavaコマンドがないことをします。はのようなものです。

- Java JREまたはJDKがまったくインストールされていないため、
- シェルファイルでPATHしくをしていないか、または
- のシェルにするファイルを「」していません。

するがあるについては、[「Javaのインストール」](#)をしてください。

"メインクラスをつけることができませんでした"

このエラーメッセージは、したエントリポイントクラスをロードできなかった、`java` コマンドによってされます。にえば、これがこりる3つのきながあります。

- しないエントリポイントクラスをしました。
- クラスはしますが、クラスをしくしていません。
- クラスがし、クラスパスをしくしましたが、クラスパスがしくないため、Javaがつかりません。

をしてするはのとおりです。

1. エントリポイントクラスのなをします。

- クラスのソースコードがある、フルネームはパッケージとなクラスでされます。 "`Main`"クラスが "`com.example.myapp`" パッケージでされたインスタンスは、が "`com.example.myapp.Main`" です。
- コンパイルされたクラスファイルをおちのは、`javap` をしてクラスをつけることができます。
- クラスファイルがディレクトリにある、ディレクトリからなクラスをすることができます。
- クラスファイルがJARファイルまたはZIPファイルの、JARファイルまたはZIPファイルのファイルパスからなクラスをできます。

2. `java` コマンドのエラーメッセージをてください。このメッセージは、`java` がしようとしているなクラスですがあります。

- エントリポイントクラスのなクラスとにすることをしてください。
- ".java" または ".class" でわるべきではありません。
- スラッシュまたは `Java1` ででないのをむべきではありません。
- のケーシングはなクラスとにするがあります。

3. しいクラスをしているは、そのクラスがにクラスパスにあることをしてください。

- クラスがマップされているパスをべてください。 [クラスからパスへのマッピングをし](#)
[てください](#)
- クラスパスがであるかをべる。のをしてください。 [クラスパスをするさまざまな](#)
- クラスパスのJARファイルとZIPファイルのそれぞれに、なパスをつクラスがまれているかどうかをします。
- ディレクトリをべて、パスがディレクトリのファイルにされるかどうかをします。

でクラスパスをしてもがつからないは、`-Xdiag` と `-XshowSettings` オプションをできます。はロードされているすべてのクラスをリストし、はJVMのクラスパスをむをします。

に、こののいくつかのながあります。

- しなないクラスをするMain-ClassをつなJARファイル。
- なClass-PathをつなJARファイル。
- クラスのに2つのオプションをさせた、javaコマンドはクラスの1つをしようとしします。
- かがJavaスタイルルールをして、だけがなるパッケージまたはクラスをし、ファイルのをでないものとしてうプラットフォームでしている。
- コードまたはコマンドラインのクラスにホモグリフにする

"メインメソッドがクラス<name>につかりません"

これは、javaコマンドがしたクラスをつけてみめるが、エントリポイントメソッドをつけることができないにする。

には3つのがあります。

- なJARファイルをしようとする、JARのマニフェストに、なエントリポイントクラスではないクラスをするな「Main-Class」があります。
- あなたは、エントリポイントクラスではないクラスをjavaコマンドにえました。
- エントリポイントクラスがしくありません。については、「[エントリポイントクラス](#)」をしてください。

そののりソース

- 「メインクラスがつかからないかロードできませんでした」とはどういうですか
- <http://docs.oracle.com/javase/tutorial/getStarted/problems/index.html>

1 - Java 8では、javaコマンドはファイルのり"/"または""をピリオド".".にマップするのにちます。ただし、これはマニュアルページにはされていません。

2 - テキストエディタがのハイフンのわりに"ハイフン"をしているされたドキュメントからコマンドをコピーペーストすると、にあいまいなケースです。

ライブラリをつJavaアプリケーションの

これは、「メインクラス」と「なJAR」ののきです。

なJavaアプリケーションは、アプリケーションのコードと、した、またはサードパーティによってされた、さまざまなライブラリコードでされています。は、にライブラリとばれ、はJARファイルとしてパッケージされています。

Javaはにバインドされたです。ライブラリをつJavaアプリケーションをする、JVMはにじてクラスをロードできるように、がどこにあるかをるがあります。して、これにするには2つのがあります。

- アプリケーションとそのは、なすべてのクラスとリソースをむのJARファイルにパッケージできます。

- JVMは、ランタイムクラスパスをしてするJARファイルをどこにつけるかをえることができます。

なJARファイルの、クラスパスは "Class-Path" マニフェストによってされます。それのは、`-cp` オプションをするか、`CLASSPATH` をして、クラスパスをするがありますこれについては、`jar` コマンドののトピックでしてください。

たとえば、エントリポイントクラスが`com.example.MyApp`の "myApp.jar" ファイルにJavaアプリケーションがあるとします。アプリケーションがライブラリJARファイル "lib / library1.jar" と "lib / library2.jar" にしているとします。コマンドラインのように`java` コマンドをしてアプリケーションをできます。

```
$ # Alternative 1 (preferred)
$ java -cp myApp.jar:lib/library1.jar:lib/library2.jar com.example.MyApp

$ # Alternative 2
$ export CLASSPATH=myApp.jar:lib/library1.jar:lib/library2.jar
$ java com.example.MyApp
```

Windowsでは、クラスパスりとして`:`わりに`;`をし`;` `export` ではなく `set` をしてローカル `CLASSPATH` を `set` ます。

Javaはそれにれていますが、「ユーザーフレンドリー」ではありません。したがって、ユーザーがののないをすために、なシェルスクリプトまたはWindowsバッチファイルをくのがなです。たとえば、のシェルスクリプトを "myApp" というファイルにれてにし、それをコマンドパスのディレクトリにくと、

```
#!/bin/bash
# The 'myApp' wrapper script

export DIR=/usr/libexec/myApp
export CLASSPATH=$DIR/myApp.jar:$DIR/lib/library1.jar:$DIR/lib/library2.jar
java com.example.MyApp
```

のようにすることができます

```
$ myApp arg1 arg2 ...
```

コマンドラインのはすべて、`"$@"` をしてJavaアプリケーションにされます。シンタックスはなりますが、Windowsバッチファイルとのことができます。

のおよびそのの

まずに、のスペースをするは、にはJavaのではありません。むしろ、Javaプログラムをするときにしているコマンドシェルによってされるがあるです。

たとえば、ファイルのサイズをするのなプログラムがあるとします。

```
import java.io.File;

public class PrintFileSizes {

    public static void main(String[] args) {
        for (String name: args) {
            File file = new File(name);
            System.out.println("Size of '" + file + "' is " + file.size());
        }
    }
}
```

パスにスペースが含まれているファイルのサイズをしたいと思います。 `/home/steve/Test File.txt`。のようなコマンドをすると

```
$ java PrintFileSizes /home/steve/Test File.txt
```

シェルは `/home/steve/Test File.txt` がにはつのパスであることをりません。わりに、2つのなるを Java アプリケーションにします。これらのはそれぞれのファイルサイズをつけようとし、それらのパスをつファイルがしないためにします。

POSIX シェルをするソリューション

POSIX シェルには `sh` や `bash` や `ksh` などのもまれています。これらのシェルのいずれかをしているは、をすることでできます。

```
$ java PrintFileSizes "/home/steve/Test File.txt"
```

パスののは、のとしてされるべきであることをシェルにえます。この、はされます。これをうにはいくつかのがあります

```
$ java PrintFileSizes '/home/steve/Test File.txt'
```

はのようにわれますが、のさまざまなもされます。

```
$ java PrintFileSizes /home/steve/Test\ File.txt
```

バックスラッシュはのスペースをエスケープし、としてされないようにします。

のののをうのをむよりなドキュメントについては、 [Bash](#) ドキュメントの [トピック](#) をしてください。

Windows ソリューション

Windows のなは、OS レベルでは、がの [ソース](#) としてプロセスにされることです。つまり、コマンドラインのまたはのなは、プログラムまたはランタイムライブラリのいずれかになります。がたくさんあります。

Javaの、いストーリーをする

- `java` コマンドののにをれることができます。これにより、にスペースをれてすことができます。
- どうやら、`java` コマンドはコマンドをしており、かれなかれしい
- しかし、これをバッチファイルで`SET`とのとみわせようとすると、がされるかどうかはにになります。
- `cmd.exe` シェルにはらかにのエスケープメカニズムがあります。をにし、`^`エスケープをうなどです。

については、[バッチファイルのドキュメント](#)をしてください。

Java オプション

`java` コマンドは、いオプションをサポートしています。

- すべてのオプションはのハイフンまたはマイナス `-` でまります。「い」オプションに`--`をするGNU / Linuxのはサポートされていません。
- オプションをさせるには、`<classname>`または`-jar <jarfile>`のにオプションをするがあります。そののは、のJavaアプリケーションにすとしてわれます。
- `-X`または`-XX`まらないオプションはオプションです。あなたは、のオプションをサポートするために、すべてのJava¹にすることができます。
- `-X`でまるオプションはオプションであり、あるJavaバージョンからのJavaバージョンへときされるがあります。
- `-XX`でまるオプションは、なオプションであり、りすこともできます。

`-D` システムプロパティの

`-D<property>=<value>` オプションをして、システム`Properties` オブジェクトのプロパティをします。このパラメータは、なるプロパティをするためにりすことができます。

メモリ、スタック、およびガベージコレクタオプション

ヒープとスタックのサイズをするためのなオプションは、[ヒープ](#)、[PermGen](#)、および[スタックのサイズ](#)のされています。ガベージコレクタのオプションはじトピックですがあります

アサーションのと

`-ea`

オプションと`-da`オプションはそれぞれ、Java `assert` チェックをまたはにします。

- すべてのアサーションチェックはデフォルトでになっています。
- `-ea` オプションをすると、すべてのアサーションをチェックでき`-ea`
- `-ea:<packagename>...`は、パッケージおよびすべてのサブパッケージのアサーションのチェックをにします。
- `-ea:<classname>...`のアサーションのチェックをにします。
- `-da` オプションは、すべてのアサーションのチェックをにします。
- `-da:<packagename>...`は、パッケージおよびすべてのサブパッケージのアサーションのチェックをにします。
- `-da:<classname>...`のアサーションのチェックをにします。
- `-esa` オプションをすると、すべてのシステムクラスをチェックできます。
- `-dsa` オプションは、すべてのシステムクラスのをにします。

オプションはみわせることができます。えは。

```
$ # Enable all assertion checking in non-system classes
$ java -ea -dsa MyApp

$ # Enable assertions for all classes in a package except for one.
$ java -ea:com.wombat.fruitbat... -da:com.wombat.fruitbat.Brickbat MyApp
```

アサーションチェックをにすると、Javaプログラミングのがされるがあることにしてください。

- にアプリケーションをよりくするがあります。
- のメソッドのにがかり、マルチスレッドアプリケーションのスレッドのタイミングがわるがあります。
- それはをさせるのあるせぬ - をするがある。
- ってされた`assert`ステートメントは、ましくないをつがあります。

VMタイプの

`-client`オプションと`-server`オプションをすると、2つのなるのHotSpot VMをできます。

- 「クライアント」フォームは、ユーザーアプリケーションにされており、よりなをします。
- 「サーバー」フォームは、されるアプリケーションにされています。JVMが「ウォームアップ」にをキャプチャするのにがかり、JITコンパイラはネイティブコードのをよりうまくえます。

デフォルトでは、JVMは、プラットフォームのにじて、であれば64ビットモードでされます。 `-d32`および`-d64`オプションをすると、モードをにできます。

1 - `java` コマンドのマニュアルをしてください。によっては、オプションが「`」`としてされることもあります。

オンラインでJavaコマンド - '`java`'と '`javaw`'をむ <https://riptutorial.com/ja/java/topic/5791/javaコマンド>

ンド-----java-と--javaw-

25: Javaコンパイラ - 'javac'

`javac` コマンドは、Javaソースファイルをバイトコードファイルにコンパイルするために使われます。バイトコードファイルはプラットフォームにしません。つまり、あるのハードウェアとオペレーティングシステムでコードをコンパイルし、Javaをサポートするのプラットフォームでコードをすることができます。

`javac` コマンドは、Java Development KitJDKディストリビューションにまれています。

JavaコンパイラとそのJavaツールチェーンでは、コードにのがあります。

- ソースコードは、".java"
- バイトコードは ".class"をつファイルでされます。
- ファイルシステムのソースおよびバイトコードファイルの、ファイルパスはパッケージおよびクラスのをすることができます。

バイトコードをネイティブコードにコンパイルする[ジャストインタイムJITコンパイラ](#)と `javac` コンパイラをしないでください。

Examples

'javac'コマンド - はじめに

な

「HelloWorld.java」にのJavaソースがまれているとします。

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

のコードのについては、[「Javaのい」](#)をしてください。

のコマンドをして、のファイルをコンパイルできます。

```
$ javac HelloWorld.java
```

これにより、"HelloWorld.class"というファイルがされ、のようにされます。

```
$ java HelloWorld
Hello world!
```

こののはのとおりです。

1. ソースファイル「HelloWorld.java」とは、ソースファイル...のクラスとするがあります HelloWorld。しない、コンパイルエラーがします。
2. バイトコードファイル "HelloWorld.class"はクラスにします。 "HelloWorld.class"のをするは、しようとするとエラーがします。
3. javaをしてJavaアプリケーションをするときは、バイトコードではなくクラスをします。

パッケージをした

もなJavaコードでは、パッケージをしてクラスのネームスペースをし、ったクラスののリスクをします。

com.exampleというパッケージ・コールでHelloWorldクラスをしたい、「HelloWorld.java」にはのJavaソースがまれます。

```
package com.example;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

このソースコードファイルは、パッケージのにするのディレクトリツリーにするがあります。

```
.    # the current directory (for this example)
|
----com
    |
    ----example
        |
        ----HelloWorld.java
```

のコマンドをして、のファイルをコンパイルできます。

```
$ javac com/example/HelloWorld.java
```

これにより、 "com / example / HelloWorld.class"というファイルがされます。つまり、コンパイル、ファイルはのようになります。

```
.    # the current directory (for this example)
|
----com
    |
    ----example
        |
        ----HelloWorld.java
        ----HelloWorld.class
```

に、のようにアプリケーションをできます。


```
$ java com.example.HelloWorld
Hello world!
```

こののはのとおりです。

1. ディレクトリはパッケージのとするがあります。
2. クラスをするときは、なクラスをするがあります。「com.example.HelloWorld」は「HelloWorld」ではありません。
3. のディレクトリからJavaコードをコンパイルしてするはありません。ここではのためだけにしています。

javacでのファイルをにコンパイルする。

アプリケーションがのソースコードファイルでされているほとんどの、に1つずつコンパイルできます。のとして、パスをしてのファイルをにコンパイルすることもできます

```
$ javac Foo.java Bar.java
```

またはコマンドシェルのファイルワイルドカードをして...

```
$ javac *.java
$ javac com/example/*.java
$ javac */**/*.java #Only works on Zsh or with globstar enabled on your shell
```

これにより、のディレクトリの "com / example"ディレクトリのすべてのJavaソースファイル、およびディレクトリのにそれぞれのJavaソースファイルがコンパイルされます。3のは、ソースファイルおよびコンパイラオプションのリストをファイルとしてすることです。えは

```
$ javac @sourcefiles
```

sourcefiles ファイルにはのものがまれます。

```
Foo.java
Bar.java
com/example/HelloWorld.java
```

このようなコードをコンパイルすることは、さな1のプロジェクトやだけのプログラムにしています。それをえると、Javaビルドツールをしてすることをおめします。また、ほとんどのプログラマーは、みコンパイラと "プロジェクト"のビルディングをするJava IDE [NetBeans](#)、[eclipse](#)、[IntelliJ IDEA](#) をします。

よくわれるjavacオプション

javac コマンドのためのいくつかのオプションがあります

- -d

オプションは、".class"ファイルのきみディレクトリをします。

- `-sourcepath` オプションは、ソースコードパスをします。
- `-cp` または `-classpath` オプションは、クラスおよびにコンパイルされたクラスをするためのパスをします。クラスパスとそののについては、クラス [パス](#) トピックをしてください。
- `-version` オプションは、コンパイラのバージョンをします。

ので、コンパイラオプションのよりなリストについてします。

のためのなりファレンス `javac` コマンドがある [ためにOracleのマニュアルページ](#) `javac`。

のバージョンの **Java** にコンパイルする

Javaプログラミングおよびそのは、のからリリースされて、くのがわれています。これらのには、

- Javaプログラミングのとセマンティクスの
- JavaクラスライブラリによってされるAPIの。
- Javaバイトコードセットとクラスファイルフォーマットの。

ほとんどなくえば、`enum` キーワード、いくつかの""クラスへのなど、これらのはがあります。

- いバージョンのJavaツールチェーンをしてコンパイルされたJavaプログラムは、コンパイルせずにしいバージョンのJavaプラットフォームでされます。
- いバージョンのJavaでかれたJavaプログラムは、しいJavaコンパイラでにコンパイルされます。

い **Java** をしいコンパイラでコンパイルする

しいJavaプラットフォームでいJavaコードをコンパイルしてしいプラットフォームであるがあるは、はなコンパイルフラグをえるはありません。いくつかのケースではえば、`enum` をとしてしたなど、`--source` オプションをしてしいをにすることができます。たとえば、のクラスをします。

```
public class OldSyntax {  
    private static int enum; // invalid in Java 5 or later  
}
```

Java 5コンパイラまたはそれをしてクラスをコンパイルするには、のものがです。

```
$ javac -source 1.4 OldSyntax.java
```

いプラットフォームにコンパイルする

いJavaプラットフォームであるためにJavaをコンパイルするがある、もなは、サポートするがあるもいバージョンのJDKをインストールし、そのJDKのコンパイラをビルドですることです。

また、よりしいJavaコンパイラでコンパイルすることもできますが、です。まずに、すべきながあります。

- コンパイルするコードは、とするJavaのバージョンではできなかったJavaをしてはなりません。
- コードは、いプラットフォームではできなかったのJavaクラス、フィールド、メソッドなどにしてはいけません。
- コードがするサードパーティライブラリも、いプラットフォームにされ、コンパイルおよびにでなければなりません。

がたされていると、`-target` オプションをしていプラットフォームのコードをコンパイルすることができます。えば、

```
$ javac -target 1.4 SomeClass.java
```

のクラスをコンパイルして、Java 1.4のJVMとのあるバイトコードをします。には、`-source` オプションはのある`-target` するため、`javac -source 1.4 ...`とじがあります。`--source`と`-target`のについては、Oracleのマニュアルをしてください。

`-source`、に`-target`や`-source`をうと、コンパイラのJDKがするクラスライブラリにしてコンパイルしていることになります。ではないは、しいバイトコードバージョンをつクラスでできますが、できないAPIにはします。は、`-bootclasspath` オプションをすることです。えば

```
$ javac -target 1.4 --bootclasspath path/to/javal.4/rt.jar SomeClass.java
```

ランタイムライブラリののセットにしてコンパイルします。コンパイルのクラスがしいライブラリにしていると、コンパイルエラーがします。

オンラインでJavaコンパイラ - 'javac'をむ [https://riptutorial.com/ja/java/topic/4478/javaコンパイラ](https://riptutorial.com/ja/java/topic/4478/javaコンパイラ----javac-)
----javac-

26: Javaソケット

き

ソケットは、マシンでしているかどうかをわす、に2つのプログラムのをするのにつレベルのネットワークインタフェースです。

ソケットプログラミングは、もくされているネットワーキングのコンセプトの1つです。

インターネットプロトコルトラフィックには2あります。

1. TCP - プロトコル 2. UDP - ユーザデータグラムプロトコル

TCPはのプロトコルです。

UDPはコネクションレスプロトコルです。

TCPはいがされるアプリケーションにしており、はではありません。

UDPは、ゲームなどのでなをとするアプリケーションにしています。UDPのステートレスなは、なのクライアントからのさなクエリにえるサーバーにもちます。

なで -

データをうことができないや、データのにがからないは、TCPをしてください。をしたり、データのがにならないは、UDPをしてください。

されたデータはそのまま、TCPのにされたのとじですというながあります。

UDPでされるメッセージやパケットがまったくかないというはありません。

Examples

シンプルな**TCP**エコーバックサーバ

たちのTCPエコーバックサーバはのスレッドになります。それはまりとしてです。あなたがったものをでします。

```
public class CAPECHOServer extends Thread{

    // This class implements server sockets. A server socket waits for requests to come
    // in over the network only when it is allowed through the local firewall
    ServerSocket serverSocket;

    public CAPECHOServer(int port, int timeout){
        try {
            // Create a new Server on specified port.
            serverSocket = new ServerSocket(port);
            // SoTimeout is basiacally the socket timeout.
            // timeout is the time until socket timeout in milliseconds
            serverSocket.setSoTimeout(timeout);
```

```

        } catch (IOException ex) {
            Logger.getLogger(CAPECHOServer.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    @Override
    public void run(){
        try {
            // We want the server to continuously accept connections
            while(!Thread.interrupted()){

            }
            // Close the server once done.
            serverSocket.close();
        } catch (IOException ex) {
            Logger.getLogger(CAPECHOServer.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}

```

すぐをける。runメソッドをしましょう。

```

@Override
public void run(){
    while(!Thread.interrupted()){
        try {
            // Log with the port number and machine ip
            Logger.getLogger((this.getClass().getName())).log(Level.INFO, "Listening for
Clients at {0} on {1}", new Object[]{serverSocket.getLocalPort(),
InetAddress.getLocalHost().getHostAddress()});
            Socket client = serverSocket.accept(); // Accept client connction
            // Now get DataInputStream and DataOutputStreams
            DataInputStream istream = new DataInputStream(client.getInputStream()); // From
client's input stream
            DataOutputStream ostream = new DataOutputStream(client.getOutputStream());
            // Important Note
            /*
                The server's input is the client's output
                The client's input is the server's output
            */
            // Send a welcome message
            ostream.writeUTF("Welcome!");

            // Close the connection
            istream.close();
            ostream.close();
            client.close();
        } catch (IOException ex) {
            Logger.getLogger(CAPECHOServer.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    // Close the server once done

    try {
        serverSocket.close();
    } catch (IOException ex) {
        Logger.getLogger(CAPECHOServer.class.getName()).log(Level.SEVERE, null, ex);
    }
}

```

```
}
```

すぐあなたがtelnetをいいてをみることができるなら、あなたはWelcomeメッセージをるでしょう。

したポートとIPアドレスにするがあります。

のようながされます。

```
Welcome!
```

```
Connection to host lost.
```

さて、たちがしたため、がわれました。によっては、のTCPクライアントをプログラムするがあります。この、ユーザーからのをしてネットワークをしてし、のをけるクライアントがです。

サーバーがにデータをする、クライアントはにデータをみるがあります。

```
public class CAPECHOCClient extends Thread{

    Socket server;
    Scanner key; // Scanner for input

    public CAPECHOCClient(String ip, int port){
        try {
            server = new Socket(ip, port);
            key = new Scanner(System.in);
        } catch (IOException ex) {
            Logger.getLogger(CAPECHOCClient.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    @Override
    public void run(){
        DataInputStream istream = null;
        DataOutputStream ostream = null;
        try {
            istream = new DataInputStream(server.getInputStream()); // Familiar lines
            ostream = new DataOutputStream(server.getOutputStream());
            System.out.println(istream.readUTF()); // Print what the server sends
            System.out.print(">");
            String tosend = key.nextLine();
            ostream.writeUTF(tosend); // Send whatever the user typed to the server
            System.out.println(istream.readUTF()); // Finally read what the server sends
            before exiting.
        } catch (IOException ex) {
            Logger.getLogger(CAPECHOCClient.class.getName()).log(Level.SEVERE, null, ex);
        } finally {
            try {
                istream.close();
                ostream.close();
                server.close();
            } catch (IOException ex) {
                Logger.getLogger(CAPECHOCClient.class.getName()).log(Level.SEVERE, null, ex);
            }
        }
    }
}
```

```
}  
}
```

すぐサーバーをする

```
ostream.writeUTF("Welcome!");  
  
String inString = istream.readUTF();    // Read what the user sent  
String outString = inString.toUpperCase(); // Change it to caps  
ostream.writeUTF(outString);  
  
// Close the connection  
istream.close();
```

そして、サーバーとクライアントをします。これにしがたです

```
Welcome!  
>
```

オンラインでJavaソケットをむ <https://riptutorial.com/ja/java/topic/9923/javaソケット>

27: Javaでの2Dグラフィックス

き

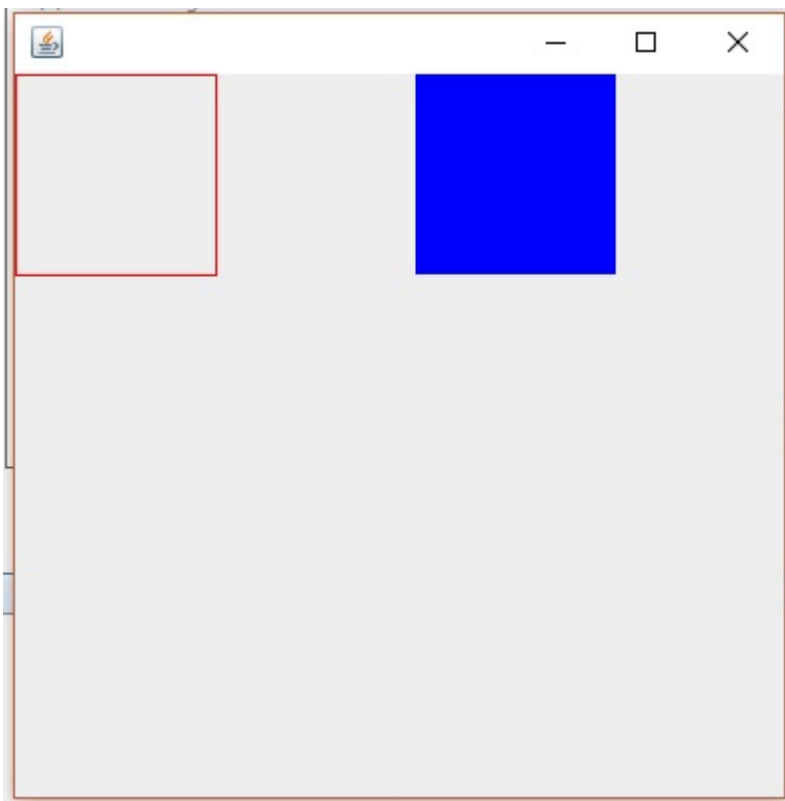
グラフィックは、キャンバス、スクリーン、またはなどのビジュアルイメージまたはデザインで、をえたり、したり、しんだりするものです。それには、コンピュータおよび、およびグラフィックアート、およびレクリエーションソフトウェアにおけるデータのがまれる。コンピュータによってされたは、コンピュータグラフィックスとされます。

Java 2D APIはです。Javaで2Dグラフィックをうにはのがあります。

Examples

1Javaをしたのとりつぶし

これは、にとりつぶしのをします。 <https://i.stack.imgur.com/dlC5v.jpg>



Graphicsクラスのほとんどのメソッドは、2つのなグループにけられます。

1. な、テキスト、イメージをレンダリングできるメソッドのとりつぶし
2. そのとりのにをえるメソッド

コードをき、そのにをりつぶすしのをあげてみましょう。ここでは、2つのクラスをします.1つのクラスはMyPanelで、のクラスはTestです。 MyPanelクラスでは、drawRectおよびfillRect methodsをしてRectangleをし、Colorをりつぶします。 setColorColor.blueメソッドでをします。

2のクラスでは、テストクラスであるグラフィックをテストします。フレームをし、p = new MyPanelオブジェクトでMyPanelをします。テストクラスをすると、RectangleとBlue Color Filled Rectangleがされます。

ファーストクラスMyPanel

```
import javax.swing.*;
import java.awt.*;
// MyPanel extends JPanel, which will eventually be placed in a JFrame
public class MyPanel extends JPanel {
    // custom painting is performed by the paintComponent method
    @Override
    public void paintComponent(Graphics g){
        // clear the previous painting
        super.paintComponent(g);
        // cast Graphics to Graphics2D
        Graphics2D g2 = (Graphics2D) g;
        g2.setColor(Color.red); // sets Graphics2D color
        // draw the rectangle
        g2.drawRect(0,0,100,100); // drawRect(x-position, y-position, width, height)
        g2.setColor(Color.blue);
        g2.fillRect(200,0,100,100); // fill new rectangle with color blue
    }
}
```

セカンドクラステスト

```
import javax.swing.*;
import java.awt.*;
public class Test { //the Class by which we display our rectangle
    JFrame f;
    MyPanel p;
    public Test(){
        f = new JFrame();
        // get the content area of Panel.
        Container c = f.getContentPane();
        // set the LayoutManager
        c.setLayout(new BorderLayout());
        p = new MyPanel();
        // add MyPanel object into container
        c.add(p);
        // set the size of the JFrame
        f.setSize(400,400);
        // make the JFrame visible
        f.setVisible(true);
        // sets close behavior; EXIT_ON_CLOSE invokes System.exit(0) on closing the JFrame
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    public static void main(String args[ ]){
        Test t = new Test();
    }
}
```

ボーダー・レイアウトの <https://docs.oracle.com/javase/tutorial/uiswing/layout/border.html>

paintComponent

- のなです
- デフォルトでは、まずをりつぶします
- その、カスタムペイント、などをし、

Graphic2Dは**Graphic2D**クラスをします

Java 2D APIをすると、のタスクをにできます。

- 、そののをきます。
- これらのをやグラデーション、テクスチャでりつぶします。
- フォントとレンダリングプロセスをかくするためのオプションきのテキストをします。
- にじてフィルタリングをして、をします。
- のレンダリングのいずれかでやなどのをします。

2のとりつぶし

```
import javax.swing.*;
import java.awt.*;

public class MyPanel extends JPanel {
    @Override
    public void paintComponent(Graphics g){
        // clear the previous painting
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D)g;
        g2.setColor(Color.blue);
        g2.drawOval(0, 0, 20,20);
        g2.fillOval(50,50,20,20);
    }
}
```

g2.drawOvalint x、int y、int height、int width;

このメソッドは、えられたさとでされたxとyのにをします。

g2.fillOvalint x、int y、int height、int width;このメソッドは、されたxおよびyのされたさおよびのをりつぶします。

オンラインでJavaでの2Dグラフィックスをむ <https://riptutorial.com/ja/java/topic/10127/javaでの2dグラフィックス>

28: Javaデプロイメント

き

Javaアプリケーション、Webアプリケーションなどを「パッケージ」して、するプラットフォームにするためのさまざまながあります。それらは、なライブラリまたはなJARファイル、WARおよびEARファイルから、インストーラおよびのファイルまであります。

もなレベルでは、コンパイルされたクラスすなわち、".class"ファイルまたはコンパイルされたクラスをむディレクトリツリーをコピーすることによって、Javaプログラムをデプロイすることができます。しかし、Javaは、のいずれかのでデプロイされます。

- JARファイルまたはJARファイルのコレクションをするシステムにJARファイルまたはJARファイルのコピーをコピーする。例えばjavacをします。
- WAR、EARなどのファイルを「サーブレット・コンテナ」または「アプリケーション・サーバー」にコピーまたはアップロードする。
- をするアプリケーションインストーラをします。インストーラーはみみJREもインストールすることがあります。
- アプリケーションのJARファイルをWebサーバーにすることで、Java WebStartをしてアプリケーションをできるようにします。

「JAR、WARおよびEARファイルの」のは、これらのファイルをするさまざまなをまとめたものです。

Javaのオープンソースおよびの「インストーラジェネレータ」と「EXEジェネレータ」ツールはくあります。に、Javaクラスファイルをするためのツールリバースエンジニアリングをよりにするためとランタイムライセンスチェックをするためのツールがあります。これらはすべて「Javaプログラミング」ドキュメントのです。

Examples

コマンドラインからなJARをる

jarをするには、1つのクラスファイルがです。これは、ダブルクリックでするは、メインメソッドがです。

このでは、のものをします。

```
import javax.swing.*;
import java.awt.Container;
```

```
public class HelloWorld {

    public static void main(String[] args) {
        JFrame f = new JFrame("Hello, World");
        JLabel label = new JLabel("Hello, World");
        Container cont = f.getContentPane();
        cont.add(label);
        f.setSize(400,100);
        f.setVisible(true);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

}
```

HelloWorld.javaというになっています

に、このプログラムをコンパイルします。

あなたはこれをしたいのプログラムをすることができます。コマンドラインからするには、の [javaプログラムのコンパイル](#) とにするドキュメントをしてください。

HelloWorld.classをしたら、しいフォルダをし、なものをびします。

manifest.txtというのファイルをしてりけます

```
Main-Class: HelloWorld
Class-Path: HelloWorld.jar
```

HelloWorld.classとしフォルダにく

コマンドラインをして、のディレクトリ `cd C:\Your\Folder\Path\Here` on Windowsをフォルダにします。

ターミナルをして、フォルダMac `cd /Users/user/Documents/Java/jarfolder` のフォルダにします。

それがしたら、 `jar -cvfm HelloWorld.jar manifest.txt HelloWorld.class` とし、 `jar -cvfm HelloWorld.jar manifest.txt HelloWorld.class` をします。これにより、されたHelloWorld.jarというの.classファイルをしてjarファイルマニフェストとHelloWorld.classがあるフォルダにあります。されます。オプションにするについては、セクションをしてください-mや-vなど。これらの、マニフェストファイルをしてディレクトリにし、HelloWorld.jarをつけます。それをクリックすると、Hello, Worldがテキストボックスにされます。

JAR、WAR、およびEARファイルの

JAR、WARおよびEARファイルのは、に "マニフェスト"ファイルとWARおよびEARファイルのディレクトリ/ファイルをつZIPファイルです。

これらのファイルをするためのされるは、それぞれのファイルタイプのを「」するJavaのビルドツールをすることです。ビルドツールをしないは、IDEの "export"がにすオプションです。

のこれらのファイルをするのは、それぞれのツールのドキュメントにされていますのでにしてく

ださい。

MavenをしたJARファイルとWARファイルの

MavenをしてJARまたはWARをすることは、にしい<packaging>をPOMファイルにれることのです。
えば、

```
<packaging>jar</packaging>
```

または

```
<packaging>war</packaging>
```

については。エントリポイントクラスとにするなをmaven jarプラグインのプラグインプロパティとしてして、""なJARファイルをするようにMavenをすることができます。アプリケーションとそののをJARファイルにする "uberJAR"ファイルをするためのプラグインもあります。

については、Mavenのドキュメント <http://www.riptinar.com/topic/898> をしてください。

AntをしたJAR、WAR、およびEARファイルの

Antビルドツールには、JAR、WAR、EARをビルドするための々の「タスク」があります。については、Antのドキュメント <http://www.riptinar.com/topic/4223> をしてください。

IDEをしたJAR、WARおよびEARファイルの

3つのもなJava IDEには、すべてデプロイメントファイルをするためのみみサポートがあります。はしばしば「エクスポート」とばれます。

- Eclipse - <http://www.riptinar.com/topic/1143>
- NetBeans - <http://www.riptinar.com/topic/5438>
- IntelliJ-IDEA - [エクスポート](#)

jar コマンドをして**JAR**、**WAR**および**EAR**ファイルをする。

jar コマンドをしてこれらのファイルを "で"することもできます。しいにしいコンポーネントファイルをむファイルツリーをみて、マニフェストファイルをし、JARファイルをするために**jar**をするだけです。

は、 **jar** コマンドTopic [JARファイルのと](#) をしてください。

Java Web Start

Oracle Javaチュートリアルでは、 [Web Start](#)をのようにまとめています。

Java Web Startソフトウェアは、ワンクリックでフルのアプリケーションをするをします。ユーザーは、のインストールをずに、なスプレッドシートプログラムやインターネットチャットクライアントなどのアプリケーションをダウンロードしてできます。

Java Web Startのそののは、きコードのサポートとプラットフォームの、およびアプリケーションキャッシュのとコードキャッシングのサポートです。

Java Web Startは、JavaWSおよびJAWSともばれます。なはのとおりです。

- [Javaチュートリアル - レッスンJava Web Start](#)
- [Java Web Startガイド](#)
- [Java Web Start FAQ](#)
- [JNLP](#)
- [javax.jnlp APIドキュメント](#)
- [Java Web Startデベロッパーサイト](#)

Web Startはレベルで、リモートWebサーバーからJARファイルとしてパックされたJavaアプリケーションをすることによってします。はのとおりです。

- アプリケーションがされるターゲットマシンののJavaインストールJREまたはJDK。Java 1.2.2がです。
 - Java 5.0では、Web StartのサポートはJRE / JDKにまれています。
 - のリリースでは、Web Startのサポートはにインストールされていました。
 - Web Startインフラストラクチャには、ユーザーがなソフトウェアをインストールするのをするWebページにめることができるJavascriptがいくつかまれています。
- ソフトウェアをホストするWebサーバーは、ターゲットマシンにアクセスでなければなりません。
- ユーザーがWebページのリンクをしてWeb Startアプリケーションをするは、のようになります。
 - のあるWebブラウザがです
 - なセキュリティでされたブラウザでは、Webブラウザのセキュリティをなうことなく、Javaのをするようにブラウザにするをえるがあります。

JNLPファイルの

のは、JNLPのなをするためのものです。

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec="1.0+" codebase="https://www.example.com/demo"
  href="demo_webstart.jnlp">
  <information>
```

```

<title>Demo</title>
<vendor>The Example.com Team</vendor>
</information>
<resources>
  <!-- Application Resources -->
  <j2se version="1.7+" href="http://java.sun.com/products/autodl/j2se"/>
  <jar href="Demo.jar" main="true"/>
</resources>
<application-desc
  name="Demo Application"
  main-class="com.example.jwsdemo.Main"
  width="300"
  height="300">
</application-desc>
<update check="background"/>
</jnlp>

```

このとおり、JNLPファイルはXMLベースであり、はすべて<jnlp>にまれています。

- `spec`は、このファイルがしているJNPLのバージョンをします。
- `codebase`は、ファイルのりの`href` URLをするためのベースURLをします。
- `href`は、このJNLPファイルのなURLをします。
- `<information>`には、タイトル、、、ヘルプ・ウェブサイトなどのメタデータがまれています。
- `<resources>`は、なJavaバージョン、OSプラットフォーム、JARファイルなど、アプリケーションのをします。
- `<application-desc>` または `<applet-desc>` は、アプリケーションのになをします。

Web サーバ—の

Webサーバは、`.jnlp`ファイルのMIMEタイプとして`application/x-java-jnlp-file`をするようにするがあります。

JNLPファイルとアプリケーションのJARファイルは、JNLPファイルでされたURLをしてできるように、Webサーバにインストールするがあります。

ウェブページからのをにする

アプリケーションをWebリンクでするは、リンクをむページをWebサーバにするがあります。

- Java Web Startがすでにユーザのマシンにインストールされているとすると、Webページにはアプリケーションをするためのリンクがまれているがあります。えは。

```
<a href="https://www.example.com/demo_webstart.jnlp">Launch the application</a>
```

- それの、ページには、ユーザ—がしているブラウザのをし、なバージョンのJavaをダウンロードしてインストールするようするスクリプトもまれているがあります。

ここでJavaをインストールすることをしたり、JNLP WebページのがするようにWebブラウザでJavaをにすることをユーザーにすことはおめしません。

コマンドラインから**Web Start**アプリケーションをする

コマンドラインからWeb Startアプリケーションをするはです。ユーザーにJava 5.0 JREまたはJDKがインストールされているとすると、これをするだけです。

```
$ javaws <url>
```

<url>はリモートサーバーのJNLPファイルのURLです。

アプリケーションとそののための**UberJAR**の

Javaアプリケーションのなは、のファイルをコピーすることによってできることです。のJava SEクラスライブラリのみにするなアプリケーションでは、コンパイルされたすべてのアプリケーションクラスをむJARファイルをするすることでこのをたします。

アプリケーションがのライブラリにしているは、それほどではありません。JARファイルをアプリケーションJARにくだけでは、のJavaクラスローダーはライブラリクラスをつけることができず、アプリケーションはしません。そのわりに、アプリケーションクラスとリソースをクラスとリソースとともにむのJARファイルをするがあります。これらは、クラスローダーがするためののとしてするがあります。

このようなJARファイルは、しばしばUberJARとされます。

"jar" コマンドをした**UberJAR**の

UberJARをするはです。はにするためにLinuxコマンドをします。コマンドはMac OSではじでなければならず、Windowsでもです。

1. ディレクトリをし、そのディレクトリにします。

```
$ mkdir tempDir
$ cd tempDir
```

2. するJARファイルごとに、アプリケーションのクラスパスにするがあるとので、 jar コマンドをしてJARをディレクトリにします。

```
$ jar -xf <path/to/file.jar>
```

これをのJARファイルにしてうと、JARのがオーバーレイされます。

3. ビルドツリーからアプリケーションクラスをディレクトリにコピーします。


```
$ cp -r path/to/classes .
```

4. ディレクトリの中からUberJARをします。

```
$ jar -cf ../myApplication.jar
```

なJARファイルを作るは、ここでするようになMANIFEST.MFをみみます。

Mavenを使ったUberJARの

あなたのプロジェクトがMavenをしてビルドされているは、"maven-assembly"または "maven-shade"プラグインをしてUberJARをすることができます。については、 [Mavenアセンブリ](#)のトピック [Maven](#)のドキュメントををしてください。

UberJARのと

UberJARののいくつかはです。

- UberJARはにできます。
- ライブラリはであるため、UberJARのライブラリをすることはできません。

さらに、なツールングをしてUberJARを作るは、JARファイルからされないライブラリクラスをすることもできます。しかし、これは、クラスによってわれます。アプリケーションでリフレクション、アノテーションなどのをするは、クラスがってされないようにするがあります。

UberJARにもいくつかのがあります。

- じをつUberJARがたくさんあるは、それぞれにのコピーがまれます。
- いくつかのオープンソースのライブラリはUberJARでの¹のためできないがあり、ライセンスをとっています。

1 - のオープンソースライブラリライセンスでは、ライブラリをすることができます。エンドユーザーのみが、ライブラリのあるバージョンをのバージョンにきえることができます。UberJARはバージョンのきえをにするがあります。

オンラインでJavaデプロイメントをむ <https://riptutorial.com/ja/java/topic/6840/javaデプロイメント>

29: Javaでのスクリプトをする

き

Javaはになですが、そのパワーはさらにすることができます。JSR223Java Specification Request 223により、スクリプトエンジン

Java Scripting APIにより、スクリプトはJavaとできます。

Scripting APIは、スクリプトとJavaののをにすることができます。スクリプトは、クラスパスにScript Engineのをたなければなりません。

デフォルトでJavaScriptECMAScriptともばれますはデフォルトでnashornによってされています。すべてのスクリプトエンジンには、すべての、メソッドがバインディングにされたスクリプトコンテキストがあります。バッファリングされたWriterにをリダイレクトし、のものにエラーをリダイレクトするのをサポートするので、のコンテキストをすることができます。

JythonやJRubyのようなくのスクリプトエンジンライブラリがあります。クラスパスにあるり、コードをすることができます。

バインディングをしてをスクリプトにすることができます。エンジンにをすることは、にはそのエンジンのみにをすることになるため、のバインディングがながあります。じタイプのすべてのエンジンでシステムとパスなどののをするがあります。その、グローバルスコープであるバインディングがです。じEngineFactoryでされたすべてのスクリプトエンジンにするをする

Examples

nashornのスクリプトモードでのjavascriptファイルの

```
public class JSEngine {

    /*
     * Note Nashorn is only available for Java-8 onwards
     * You can use rhino from ScriptEngineManager.getEngineByName("js");
     */

    ScriptEngine engine;
    ScriptContext context;
    public Bindings scope;

    // Initialize the Engine from its factory in scripting mode
    public JSEngine(){
        engine = new NashornScriptEngineFactory().getScriptEngine("-scripting");
        // Script context is an interface so we need an implementation of it
        context = new SimpleScriptContext();
        // Create bindings to expose variables into
        scope = engine.createBindings();
    }
}
```

```

// Clear the bindings to remove the previous variables
public void newBatch(){
    scope.clear();
}

public void execute(String file){
    try {
        // Get a buffered reader for input
        BufferedReader br = new BufferedReader(new FileReader(file));
        // Evaluate code, with input as bufferedReader
        engine.eval(br);
    } catch (FileNotFoundException ex) {
        Logger.getLogger(JSEngine.class.getName()).log(Level.SEVERE, null, ex);
    } catch (ScriptException ex) {
        // Script Exception is basically when there is an error in script
        Logger.getLogger(JSEngine.class.getName()).log(Level.SEVERE, null, ex);
    }
}

public void eval(String code){
    try {
        // Engine.eval basically treats any string as a line of code and evaluates it,
        // executes it
        engine.eval(code);
    } catch (ScriptException ex) {
        // Script Exception is basically when there is an error in script
        Logger.getLogger(JSEngine.class.getName()).log(Level.SEVERE, null, ex);
    }
}

// Apply the bindings to the context and set the engine's default context
public void startBatch(int SCP){
    context.setBindings(scope, SCP);
    engine.setContext(context);
}

// We use the invocable interface to access methods from the script
// Invocable is an optional interface, please check if your engine implements it
public Invocable invocable(){
    return (Invocable)engine;
}
}

```

のな

```

public static void main(String[] args) {
    JSEngine jse = new JSEngine();
    // Create a new batch probably unnecessary
    jse.newBatch();
    // Expose variable x into script with value of hello world
    jse.scope.put("x", "hello world");
    // Apply the bindings and start the batch
    jse.startBatch(ScriptContext.ENGINE_SCOPE);
    // Evaluate the code
    jse.eval("print(x);");
}

```

あなたのはこれにているはずです

```
hello world
```

あなたがることができるように、`x`がされています。すぐファイルをテストします。

ここには`test.js`があります

```
print(x);  
function test(){  
    print("hello test.js:test");  
}  
test();
```

そして、された`main`メソッド

```
public static void main(String[] args) {  
    JSEngine jse = new JSEngine();  
    // Create a new batch probably unnecessary  
    jse.newBatch();  
    // Expose variable x into script with value of hello world  
    jse.scope.put("x", "hello world");  
    // Apply the bindings and start the batch  
    jse.startBatch(ScriptContext.ENGINE_SCOPE);  
    // Evaluate the code  
    jse.execute("./test.js");  
}
```

`test.js`がアプリケーションとじディレクトリにあるとすると、このようになれるはず

```
hello world  
hello test.js:test
```

オンラインでJavaでのスクリプトをするをむ <https://riptutorial.com/ja/java/topic/9926/javaでのスクリプトをする>

30: Java ネイティブアクセス

Examples

JNA

JNA とはですか

Java Native Access(JNA)は、JavaプログラムがネイティブライブラリWindowsの.dllファイル、Unixの.soファイル...にアクセスできるコミュニティライブラリです。

どうすればえますか

- に、 [JNAのリリース](#)をダウンロードし、プロジェクトのCLASSPATHにjna.jarをします。
- に、のJavaコードをコピー、コンパイル、します。

こののでは、しているネイティブプラットフォームがWindowsであるとしします。のプラットフォームでしているは、のコードで"msvcrt"を"c"にきえてください。

のさなJavaプログラムは、 [Cの printf](#)をびすことで、コンソールにメッセージをします。

CRuntimeLibrary.java

```
package jna.introduction;

import com.sun.jna.Library;
import com.sun.jna.Native;

// We declare the printf function we need and the library containing it (msvcrt)...
public interface CRuntimeLibrary extends Library {

    CRuntimeLibrary INSTANCE =
        (CRuntimeLibrary) Native.loadLibrary("msvcrt", CRuntimeLibrary.class);

    void printf(String format, Object... args);
}
```

MyFirstJNAProgram.java

```
package jna.introduction;

// Now we call the printf function...
public class MyFirstJNAProgram {
    public static void main(String args[]) {
        CRuntimeLibrary.INSTANCE.printf("Hello World from JNA !");
    }
}
```

```
}  
}
```

どこにく

ここにのトピックにジャンプするか、 [サイト](#)にジャンプしてください。

オンラインでJavaネイティブアクセスをむ <https://riptutorial.com/ja/java/topic/5244/javaネイティブアクセス>

31: Java ネイティブインタフェース

パラメーター

パラメータ	
JNIEnv	JNIへのポインタ
	static native メソッドをびしたオブジェクト
jclass	static native メソッドをびしたクラス

JNIをするには、Javaコンパイラとネイティブコンパイラのがです。IDEとOSにじて、いくつかのがです。Eclipseのためのガイドは[ここにあります](#)。なチュートリアルは[ここで](#)つけることができます。

これらは、WindowsでJava-C ++リンケージをするです。

- `javac`をしてJavaソースファイル `.java` をクラス `.class` にコンパイルします。
- `javah`をして `native` メソッドをむJavaクラスからヘッダー `.h` ファイルをします。これらのファイルは、ネイティブコードをどのメソッドにするかを「」します。
- `native`メソッドをするC ++ソースファイル `.cpp` にヘッダファイル `#include` をインクルードします。
- C ++ソースファイルをコンパイルし、ライブラリ `.dll` をします。このライブラリにはネイティブコードのがまれています。
- ライブラリパス `-Djava.library.path` をし、Javaソースファイル `System.loadLibrary(...)` にロードします。

コールバックネイティブコードからJavaメソッドをびすでは、メソッドをするがあります。がつていと、エラーがします。このため、`javap -s`つてをするとです。

Examples

JavaからC ++メソッドをびす

Javaのメソッドとメンバーメソッドをネイティブとしてマークして、そのがライブラリファイルにあることをすことができます。ネイティブメソッドをすると、JVMはロードされたライブラリ [ネイティブライブラリのみみ](#)をでするをし、なのマングリングスキームをしてのとスタックのをい、ネイティブコードにをします。

Javaコード

```

/** com/example/jni/JNIJava.java */

package com.example.jni;

public class JNIJava {
    static {
        System.loadLibrary("libJNI_CPP");
    }

    // Obviously, native methods may not have a body defined in Java
    public native void printString(String name);
    public static native double average(int[] nums);

    public static void main(final String[] args) {
        JNIJava jniJava = new JNIJava();
        jniJava.printString("Invoked C++ 'printString' from Java");

        double d = average(new int[]{1, 2, 3, 4, 7});
        System.out.println("Got result from C++ 'average': " + d);
    }
}

```

C++コード

ネイティブをむヘッダーファイルは、クラスの `javah` ツールをしてするがあります。ビルドディレクトリでのコマンドをします。

```
javah -o com_example_jni_JNIJava.hpp com.example.jni.JNIJava
```

...のヘッダーファイルをします にするためにコメントはされています。

```

// com_example_jni_JNIJava.hpp

/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h> // The JNI API declarations

#ifdef __cplusplus
extern "C" { // This is absolutely required if using a C++ compiler
#endif

JNIEXPORT void JNICALL Java_com_example_jni_JNIJava_printString
    (JNIEnv *, jobject, jstring);

JNIEXPORT jdouble JNICALL Java_com_example_jni_JNIJava_average
    (JNIEnv *, jclass, jintArray);

#ifdef __cplusplus
}
#endif

```

にをします。


```
// com_example_jni_JNIJava.cpp

#include <iostream>
#include "com_example_jni_JNIJava.hpp"

using namespace std;

JNIEXPORT void JNICALL Java_com_example_jni_JNIJava_printString(JNIEnv *env, jobject jthis,
jstring string) {
    const char *stringInC = env->GetStringUTFChars(string, NULL);
    if (NULL == stringInC)
        return;
    cout << stringInC << endl;
    env->ReleaseStringUTFChars(string, stringInC);
}

JNIEXPORT jdouble JNICALL Java_com_example_jni_JNIJava_average(JNIEnv *env, jclass jthis,
jintArray intArray) {
    jint *intArrayInC = env->GetIntArrayElements(intArray, NULL);
    if (NULL == intArrayInC)
        return -1;
    jsize length = env->GetArrayLength(intArray);
    int sum = 0;
    for (int i = 0; i < length; i++) {
        sum += intArrayInC[i];
    }
    env->ReleaseIntArrayElements(intArray, intArrayInC, 0);
    return (double) sum / length;
}
```

ののクラスをすると、のようになられます。

JavaからびされたC ++ 'printString'
C ++ 'average'のがられました3.4

C ++ コールバックからJava メソッドをびす

ネイティブコードからJavaメソッドをびすことは、2つのステップからなるプロセスです。

1. メソッドとをして、 `GetMethodID` JNIをつメソッドポインタをします。
2. [ここにリストされている](#) `Call*Method` の1つを `Call*Method` [ます](#)。

Javaコード

```
/** com.example.jni.JNIJavaCallback.java */

package com.example.jni;

public class JNIJavaCallback {
    static {
        System.loadLibrary("libJNI_CPP");
    }

    public static void main(String[] args) {
        new JNIJavaCallback().callback();
    }
}
```

```

}

public native void callback();

public static void printNum(int i) {
    System.out.println("Got int from C++: " + i);
}

public void printFloat(float i) {
    System.out.println("Got float from C++: " + i);
}
}

```

C ++コード

```

// com_example_jni_JNICppCallback.cpp

#include <iostream>
#include "com_example_jni_JNIJavaCallback.h"

using namespace std;

JNIEXPORT void JNICALL Java_com_example_jni_JNIJavaCallback_callback(JNIEnv *env, jobject
jthis) {
    jclass thisClass = env->GetObjectClass(jthis);

    jmethodID printFloat = env->GetMethodID(thisClass, "printFloat", "(F)V");
    if (NULL == printFloat)
        return;
    env->CallVoidMethod(jthis, printFloat, 5.221);

    jmethodID staticPrintInt = env->GetStaticMethodID(thisClass, "printNum", "(I)V");
    if (NULL == staticPrintInt)
        return;
    env->CallVoidMethod(jthis, staticPrintInt, 17);
}

```

C ++からの5.221

C ++からintをました17

ディスクリプタの

またはシグネチャは、コンパイルされた.classファイルのjavapプログラムをしてされます。

javap -p -s com.example.jni.JNIJavaCallbackをにします。

```

Compiled from "JNIJavaCallback.java"
public class com.example.jni.JNIJavaCallback {
    static {};
    descriptor: ()V

    public com.example.jni.JNIJavaCallback();
    descriptor: ()V

    public static void main(java.lang.String[]);

```

```

    descriptor: ([Ljava/lang/String;)V

    public native void callback();
        descriptor: ()V

    public static void printNum(int);
        descriptor: (I)V // <---- Needed

    public void printFloat(float);
        descriptor: (F)V // <---- Needed
}

```

ネイティブライブラリをロードする

ライブラリファイルをJavaでロードするためのなイディオムは、のとおりです。

```

public class ClassWithNativeMethods {
    static {
        System.loadLibrary("Example");
    }

    public native void someNativeMethod(String arg);
    ...
}

```

`System.loadLibrary`への`System.loadLibrary`は、ほとんどの、クラスのロードにするようにであり、ライブラリがロードされるにネイティブメソッドがされないようにします。しかし、のことがです

```

public class ClassWithNativeMethods {
    // Call this before using any native method
    public static void prepareNativeMethods() {
        System.loadLibrary("Example");
    }

    ...
}

```

これにより、になるまでライブラリのみみをらせることができますが、`java.lang.UnsatisfiedLinkError`をけるためにはながです。

ターゲットファイルの

ライブラリファイルは、`java.library.path`システムプロパティでされたパスでされます。に
`Djava.library.path=JVM`をしてきすることができます。

```
java -Djava.library.path=path/to/lib/:path/to/other/lib MainClassWithNativeMethods
```

システムのパスりにしてください;わりに:。

`System.loadLibrary`は、プラットフォームのでライブラリファイルをすることにしてください。の
コードスニペットは、Linuxでは`libExample.so`、Windowsでは`Example.dll`というのファイルが
`libExample.so`

です。

`System.loadLibrary`のわりに`System.loadLibrary` `System.load(String)`があります。これはライブラリファイルへのパスをとり、`java.library.path`をします

```
public class ClassWithNativeMethods {  
    static {  
        System.load("/path/to/lib/libExample.so");  
    }  
  
    ...  
}
```

オンラインでJavaネイティブインタフェースをむ <https://riptutorial.com/ja/java/topic/168/javaネイティブインタフェース>

32: JavaのJSON

き

JSONJavaScript Object Notationは、やがみきするのにな、でテキストベースのにしないデータです。JSONは、オブジェクトとという2つのをできます。JSONは、Ajaxアプリケーション、データベース、およびRESTful Webサービスでよくされます。JSONのJava APIは、JSONを、およびするためのAPIをします。

ここでは、Google Gsonライブラリ、Jackson Object Mapperなどのさまざまなライブラリをして、JavaでJSONをおよびするをにします。

のライブラリをしたは、ここでつけることができます JSONをJavaでする

Examples

データをJSONでエンコードする

JSONObjectをしてデータをするがあるは、のをしてください。

```
// Create a new javax.json.JSONObject instance.
JSONObject first = new JSONObject();

first.put("foo", "bar");
first.put("temperature", 21.5);
first.put("year", 2016);

// Add a second object.
JSONObject second = new JSONObject();
second.put("Hello", "world");
first.put("message", second);

// Create a new JSONArray with some values
JSONArray someMonths = new JSONArray(new String[] { "January", "February" });
someMonths.put("March");
// Add another month as the fifth element, leaving the 4th element unset.
someMonths.put(4, "May");

// Add the array to our object
object.put("months", someMonths);

// Encode
String json = object.toString();

// An exercise for the reader: Add pretty-printing!
/* {
    "foo": "bar",
    "temperature": 21.5,
    "year": 2016,
    "message": {"Hello": "world"},
    "months": ["January", "February", "March", null, "May"]
}
```

```
}  
*/
```

JSONデータのデコード

JSONObject からデータをするがあるは、のをしてください。

```
String json =  
"{\"foo\": \"bar\", \"temperature\": 21.5, \"year\": 2016, \"message\": {\"Hello\": \"world\"}, \"months\": [\"J  
  
// Decode the JSON-encoded string  
JSONObject object = new JSONObject(json);  
  
// Retrieve some values  
String foo = object.getString("foo");  
double temperature = object.getDouble("temperature");  
int year = object.getInt("year");  
  
// Retrieve another object  
JSONObject secondary = object.getJSONObject("message");  
String world = secondary.getString("Hello");  
  
// Retrieve an array  
JSONArray someMonths = object.getJSONArray("months");  
// Get some values from the array  
int nMonths = someMonths.length();  
String february = someMonths.getString(1);
```

optXXXとgetXXXのメソッド

JSONObject と JSONArray は、しようとしているがしないか、ののをうになメソッドがいくつかあります。

```
JSONObject obj = new JSONObject();  
obj.putString("foo", "bar");  
  
// For existing properties of the correct type, there is no difference  
obj.getString("foo"); // returns "bar"  
obj.optString("foo"); // returns "bar"  
obj.optString("foo", "tux"); // returns "bar"  
  
// However, if a value cannot be coerced to the required type, the behavior differs  
obj.getInt("foo"); // throws JSONException  
obj.optInt("foo"); // returns 0  
obj.optInt("foo", 123); // returns 123  
  
// Same if a property does not exist  
obj.getString("undefined"); // throws JSONException  
obj.optString("undefined"); // returns ""  
obj.optString("undefined", "tux"); // returns "tux"
```

じルールがされますにgetXXX / optXXXのJSONArray。

JSONへのオブジェクトGsonライブラリ

Person nameクラスがあるとします。

```
private class Person {  
    public String name;  
  
    public Person(String name) {  
        this.name = name;  
    }  
}
```

コード

```
Gson g = new Gson();  
  
Person person = new Person("John");  
System.out.println(g.toJson(person)); // {"name":"John"}
```

もちろん、Gsonのはクラスパスになければなりません。

JSON To ObjectGsonライブラリ

Person nameクラスがあるとします。

```
private class Person {  
    public String name;  
  
    public Person(String name) {  
        this.name = name;  
    }  
}
```

コード

```
Gson gson = new Gson();  
String json = "{\"name\": \"John\"}";  
  
Person person = gson.fromJson(json, Person.class);  
System.out.println(person.name); //John
```

クラスパスにgsonライブラリがです。

JSONからをする

```
String json = "{\"name\": \"John\", \"age\":21}";  
  
JsonObject jsonObject = new JsonParser().parse(json).getAsJsonObject();  
  
System.out.println(jsonObject.get("name").getString()); //John  
System.out.println(jsonObject.get("age").getInt()); //21
```

Jackson Object Mapperの

ポホモデル

```
public class Model {
    private String firstName;
    private String lastName;
    private int age;
    /* Getters and setters not shown for brevity */
}
```

オブジェクトへの

```
Model outputObject = objectMapper.readValue(
    "{ \"firstName\": \"John\", \"lastName\": \"Doe\", \"age\": 23 }",
    Model.class);
System.out.println(outputObject.getFirstName());
//result: John
```

Object to String

```
String jsonString = objectMapper.writeValueAsString(inputObject);
//result: {"firstName":"John","lastName":"Doe","age":23}
```

なインポートステートメント

```
import com.fasterxml.jackson.databind.ObjectMapper;
```

Mavenjackson-databind

ObjectMapper インスタンス

```
//creating one
ObjectMapper objectMapper = new ObjectMapper();
```

- ObjectMapperはスレッドセーフです
- されたインスタンスをつ

デシリアライゼーション

```
<T> T readValue(String content, Class<T> valueType)
```

- valueTypeをするがあります。りはこのになります
- スロー
 - IOException - レベルのI/Oの
 - JsonParseException

- となるになコンテンツがまれている

- `JsonMappingException` - JSONがオブジェクトとしない

jsonStringはです

```
Model fromJson = objectMapper.readValue(jsonString, Model.class);
```

シリアライズの

String writeValueAsStringオブジェクト

- スロー

- `JsonProcessingException`エラーがした
- バージョン2.1よりのリリースでは、`IOException`がスローされました。2.1はそれを行いました。

JSON

`JSONObject` プロパティをりしする

```
JSONObject obj = new JSONObject("{\"isMarried\":\"true\", \"name\":\"Nikita\", \"age\":\"30\"}");
Iterator<String> keys = obj.keys();//all keys: isMarried, name & age
while (keys.hasNext()) { //as long as there is another key
    String key = keys.next(); //get next key
    Object value = obj.get(key); //get next value by key
    System.out.println(key + " : " + value);//print key : value
}
```

`JSONArray`を`JSONArray`する

```
JSONArray arr = new JSONArray(); //Initialize an empty array
//push (append) some values in:
arr.put("Stack");
arr.put("Over");
arr.put("Flow");
for (int i = 0; i < arr.length(); i++) { //iterate over all values
    Object value = arr.get(i); //get value
    System.out.println(value); //print each value
}
```

JSON Builder - メソッド

`JSONObject`と`JSONArray`しながらメソッドをできます。

`JSONObject`の

```
JSONObject obj = new JSONObject();//Initialize an empty JSON object
//Before: {}
```

```
obj.put("name","Nikita").put("age","30").put("isMarried","true");  
//After: {"name":"Nikita","age":30,"isMarried":true}
```

JSONArray

```
JSONArray arr = new JSONArray();//Initialize an empty array  
//Before: []  
arr.put("Stack").put("Over").put("Flow");  
//After: ["Stack","Over","Flow"]
```

JSONObject.NULL

`null` をプロパティにするがあるは、されたな `JSONObject.NULL` をし、の `Java null` はしないでください。

`JSONObject.NULL` は、がのプロパティをにするためにされるセンチネルです。

```
JSONObject obj = new JSONObject();  
obj.put("some", JSONObject.NULL); //Creates: {"some":null}  
System.out.println(obj.get("some")); //prints: null
```

```
JSONObject.NULL.equals(null); //returns true
```

`Java.equals()` のなはどれですか

`null` の `x` の、`x.equalsnull` は `false` をすがあります。

JSONArray から Java リストへ Gson ライブラリ

ここでは、`Java ArrayList` にしたいな `JSONArray` をします

```
{  
  "list": [  
    "Test_String_1",  
    "Test_String_2"  
  ]  
}
```

に、`JSONArray 'list'` を、する `Java ArrayList` をすのメソッドにします。

```
public ArrayList<String> getListString(String jsonList){  
    Type listType = new TypeToken<List<String>>() {}.getType();  
    //make sure the name 'list' matches the name of 'JSONArray' in your 'Json'.  
    ArrayList<String> list = new Gson().fromJson(jsonList, listType);  
    return list;  
}
```

`POM.xml` ファイルにの `Maven` をするがあります。

```
<!-- https://mvnrepository.com/artifact/com.google.code.gson/gson -->
<dependency>
  <groupId>com.google.code.gson</groupId>
  <artifactId>gson</artifactId>
  <version>2.7</version>
</dependency>
```

または、クラスパスに `com.google.code.gson:gson:jar:<version>` があるはずです。

JacksonをしてJSONコレクションをオブジェクトコレクションにデシリアライズする

あなたが `pojo` クラス `Person` をっているとします

```
public class Person {
    public String name;

    public Person(String name) {
        this.name = name;
    }
}
```

JSONや `Person` オブジェクトのマッピングにしたいとします。このため、に `List<Person>` および `Map<String, Person>` クラスをすることはできませんしたがって、それらをしてJSONをシリアルします。このをするために、 `TypeReference` `TypeFactory` と `TypeReference` 2つのアプローチをします。

TypeFactory

ここでりけるアプローチは、ファクトリおよびそのユーティリティをしてをすることです。パラメータは、するコレクションリスト、セットなどと、そのコレクションにするクラスです。

のアプローチは、あなたのをしだけし、よりきれいにえるのです。 `TypeReference` は、タイプパラメータをけります。ここで、のタイプの `List<Person>` をします。この `TypeReference` オブジェクトをインスタンスし、それをコンテナとしてするだけです。

にJSONをにJavaオブジェクトにシリアルするをてみましょう。JSONがとしてされているは、リストとしてシリアルできます。よりなネストされたがあるは、マッピングにデシリアライズするがあります。のをてみましょう。

JSONのデシリアライズ

```
String jsonString = "[{\"name\": \"Alice\"}, {\"name\": \"Bob\"}]\"
```

タイプファクトリーアプローチ

```
CollectionType listType =
```

```
factory.constructCollectionType(List.class, Person.class);
List<Preson> list = mapper.readValue(jsonString, listType);
```

タイプアプローチ

```
TypeReference<Person> listType = new TypeReference<List<Person>>() {};  
List<Person> list = mapper.readValue(jsonString, listType);
```

JSONマップをシリアルする

```
String jsonString = "{\"0\": {\"name\": \"Alice\"}, \"1\": {\"name\": \"Bob\"}}"
```

タイプファクトリーアプローチ

```
CollectionType mapType =  
    factory.constructMapLikeType(Map.class, String.class, Person.class);  
List<Person> list = mapper.readValue(jsonString, mapType);
```

タイプアプローチ

```
TypeReference<Person> mapType = new TypeReference<Map<String, Person>>() {};  
Map<String, Person> list = mapper.readValue(jsonString, mapType);
```

されるインポートステートメント

```
import com.fasterxml.jackson.core.type.TypeReference;  
import com.fasterxml.jackson.databind.ObjectMapper;  
import com.fasterxml.jackson.databind.type.CollectionType;
```

されるインスタンス

```
ObjectMapper mapper = new ObjectMapper();  
TypeFactory factory = mapper.getTypeFactory();
```

TypeReference アプローチはくえるかもしれませんが、いくつかのがあります

1. TypeReferenceはクラスをしてインスタンスするがあります
2. あなたはなをするべきです

そうしないとジェネリックがわれ、デシリアライゼーションにします。

オンラインでJavaのJSONをむ <https://riptutorial.com/ja/java/topic/840/javaのjson>

33: Javaのとし - NullとNullPointerException

`null`は、がであるフィールドのされていないのデフォルトです。

`NullPointerException` またはNPEは、 `null` オブジェクトにしてなをしようとするとスローされるです。そのようなには、

- `null` ターゲットオブジェクトでインスタンスメソッドをびす、
- `null` ターゲット・オブジェクトのフィールドにアクセスし、
- `null` オブジェクトをけしようとするか、そのさにアクセスしようとしたが、
- `synchronized` ブロックの`null` オブジェクトをミューテックスとしてして、
- `null` オブジェクトをキャストし、
- `null` オブジェクトをアンボックスする
- `null` オブジェクトをスローし`null`。

NPEのもな

- をつフィールドをするのをれると、
- ののをするのをれる
- ので`null` をすようにされたのAPIメソッドのをテストしません。

`null` をすにされるメソッドのには、

- Map APIの`get(key)` メソッドは、マッピングをたないキーでびすと`null` をし`null`。
- `ClassLoader` およびClass APIの`getResource(path)` および`getResourceAsStream(path)` メソッドは、リソースがつからないは`null` をし`null`。
- ガベージコレクタがをクリアした、Reference APIの`get()` メソッドは`null` をし`null`。
- しないリクエストパラメータ、セッションまたはセッションなどをしようとすると、Java EEサーブレットAPIのさまざまな`getXxxx` メソッドは`null` をし`null`。

`null` をにテストするか、「Yoda Notation」をするなど、ましくないNPEをするためのがあります。これらのでは、にするのあるをすというましくないがじることがあります。

Examples

とし。プリミティブラッパーをにすると、`NullPointerException`がするがあります。

しいJavaのプログラマーは、プリミティブとラッパーをじですることがあります。これはをきこすがあります。のをえてみましょう。

```
public class MyRecord {
    public int a, b;
    public Integer c, d;
}
```

```
...
MyRecord record = new MyRecord();
record.a = 1;           // OK
record.b = record.b + 1; // OK
record.c = 1;           // OK
record.d = record.d + 1; // throws a NullPointerException
```

MyRecord クラス¹では、デフォルトの値をしてフィールドの値をしています。したがって、レコードを new すると、a および b フィールドはゼロにされ、c および d フィールドは null にされ、null。

デフォルトのされたフィールドをしようとする、int フィールドはにすることがわかりますが、Integer フィールドはしないもあります。には、した d は、のが null をアンボックスしようとし、それがで NullPointerException がスローされます。

これをやるにはいくつかのがあります

- フィールド c と d がプリミティブラッパーであるがあるは、デフォルトの値にすべきではないか、または null テストするがあり null。は、null のフィールドにながないり、しいアプローチです。
- フィールドがプリミティブラッパーであるがないは、プリミティブラッパーにするのはいいです。このにえて、プリミティブラッパーはプリミティブにべてのオーバーヘッドをちます。

ここでののは、になをき、ラッパーをしないことです。

1 - このクラスはいコーディングのではありません。えは、うまくされたクラスはパブリックフィールドをたないでしよう。しかし、これはこののポイントではありません。

Pitfall - のやコレクションをすために null をう

いくつかのプログラマーは、のやコレクションをすために null をしてスペースをすることをおめします。のスペースをできるのはいいありませんが、コードがでれやすくなるということです。をすの2つのバージョンをす

のバージョンは、メソッドをどおりにコーディングするです。

```
/**
 * Sum the values in an array of integers.
 * @arg values the array to be summed
 * @return the sum
 */
public int sum(int[] values) {
    int sum = 0;
    for (int value : values) {
        sum += value;
    }
    return sum;
}
```

2のバージョンは、null をしてのをすがあるに、メソッドをコーディングするがあるです。

```
/**
 * Sum the values in an array of integers.
 * @arg values the array to be summed, or null.
 * @return the sum, or zero if the array is null.
 */
public int sum(int[] values) {
    int sum = 0;
    if (values != null) {
        for (int value : values) {
            sum += value;
        }
    }
    return sum;
}
```

このとおり、コードはもうしです。これは、このように`null`をするというにします。

に、`null`があるこのがくのでされているかどうかをします。あなたがそれをするで、`null`をテストするがあるかどうかをするがあり`null`。あなたがそこにするのある`null`をした、あなたは`NullPointerException`をにさらします。したがって、このように`null`をすると、アプリケーションがよりになります。プログラマのエラーのをよりけやすくなります。

ここでののは、のとのリストをしてそれがするときです。

```
int[] values = new int[0]; // always empty
List<Integer> list = new ArrayList(); // initially empty
List<Integer> list = Collections.emptyList(); // always empty
```

スペースのオーバーヘッドはさく、これはのあることであればにえるのがあります。

とし - せぬをる

`StackOverflow`では、`Answers`にのようなコードがよくされます。

```
public String joinStrings(String a, String b) {
    if (a == null) {
        a = "";
    }
    if (b == null) {
        b = "";
    }
    return a + ": " + b;
}
```

しばしば、これは、`NullPointerException`をするために、このような`null`をテストする "ベストプラクティス"であるアサーションをい`null`。

それはベストプラクティスですかするにいいえ。

`joinStrings`これをうことをおめするに、するがあるいくつかのながあります

"a"または "b"がnullであるとはどういうですか

`String`はゼロのである場合があります。したがって、すでにのをすがあります。 `null`は""とはかうですかいえのは、のをす2つのがあることがになります。

nullはされていないからたのですか

`null`は、されていないフィールド、またはされていないからるがあり `null`。はによって、またはによってされないことがあります。それがだった、これはバグです。

ヌルは「らない」か「」をしていますか

ときには `null` がつこともあり `null`。えは、ののがであるか、できないか、または「オプション」であるかどうか。Java 8では、`Optional` クラスはそれをするよりいをしします。

これがバグまたはミスであれば、「い」ものにすることがありますか

コードのの1つは、のをそのにすることによってせぬ `null` 「くする」ことです。しいですか `NullPointerException` するようにして、をスタックの `NullPointerException` キャッチしてバグとしてするがいでしょうか

「うまくいく」というは、をすか、するのがしくなることです。

これはコードにかいか

して "make good" アプローチをすると、コードにはくの "な"ヌルテストがまれます。これは、むのをよりくし、よりにするでしょう。さらに、このテストと「うまくいく」のすべてが、アプリケーションのパフォーマンスにするがあります。

すれば

`null` があるである、 `null` のテストがしいです。として、 `null` がつ、 `null` をけれるかすメソッドのjavadocでこれをにすることがあり `null`。

それのは、しない `null` をプログラミングエラーとしてい、がコードにがあることがわかるように `NullPointerException` させることをおめします。

Pitfall - をスローするわりにnullをす

Javaプログラマのには、をげたりしたりするというなをっているものがあります。これにより、のようなコードがされます。

```
public Reader getReader(String pathname) {
    try {
        return new BufferedReader(FileReader(pathname));
    } catch (IOException ex) {
        System.out.println("Open failed: " + ex.getMessage());
        return null;
    }
}
```

}

では、それがなのですか

は、`getReader`が`Reader`オープンできなかったことをするために`null`をなとしてすことです。りは、するに`null`かどうかをべるがあり`null`。テストがされた、は`NullPointerException`ます。

には3つのがあります。

1. あまりにもく`IOException`がキャッチされました。
2. このコードのは、リソースをらすリスクがあることをします。
3. 「の」`Reader`がされなかったため、`null`がされました。

、をこのようににらえるがあるとすると、`null`をすがいくつかありました。

1. `NullReader`クラスをすることはです。あたかもリーダーがすでに「ファイルのわり」のにあるかのようにAPIのがるうようなものです。
2. Java 8では、`getReader`を`Optional<Reader>`をすようにすることができます。

とし - **II/O** ストリームをじたときに **II/O** ストリームがされていないかどうかをチェックしない

メモリリークをぐために、ストリームまたはストリームをじることをれないでください。これは、`catch`のない`try - catch - finally`でいます。

```
void writeNullBytesToFile(int count, String filename) throws IOException {
    FileOutputStream out = null;
    try {
        out = new FileOutputStream(filename);
        for(; count > 0; count--)
            out.write(0);
    } finally {
        out.close();
    }
}
```

のコードはにえるかもしれませんが、デバッグをにするがあります。 `out` が `out` た `out = new FileOutputStream(filename)` がをスローすると、 `out.close()` がさ `out` と `out` が `null` になり、`NullPointerException` ます。

これをぐには、ストリームをじるにストリームが`null`でないことをするだけです。

```
void writeNullBytesToAFile(int count, String filename) throws IOException {
    FileOutputStream out = null;
    try {
        out = new FileOutputStream(filename);
        for(; count > 0; count--)
            out.write(0);
    } finally {
        if (out != null)
            out.close();
    }
}
```

よりいいアプローチは、リソースを `try-with-resources` で `try` ことです。これは、`finally` ブロックをとせずに `NPE` を上げるために `0` のでストリームをにじます。

```
void writeNullBytesToAFile(int count, String filename) throws IOException {
    try (FileOutputStream out = new FileOutputStream(filename)) {
        for(; count > 0; count--)
            out.write(0);
    }
}
```

とし - **`NullPointerException`** をけるために **"Yoda"** をう

`StackOverflow` にされたサンプルコードのくにはのようなスニペットがまれています

```
if ("A".equals(someString)) {
    // do something
}
```

`someString` が `null`、`NullPointerException` するのを「」または「」し `null`。さらに、

```
"A".equals(someString)
```

よりいい

```
someString != null && someString.equals("A")
```

よりで、によってはよりかもしれませんが、でするように、さはできます。

しかし、のとしは、のとして **`NullPointerExceptions`** をけるために **Yoda** テストをしています。

`"A".equals(someString)` とくと `someString` が `null` になるケースがには `someString` ようになり `null`。しかし、の **Pitfall - "しない"** "い" "い" をるがするように、"い" `null` をることはいろいろなでながあります。

これは、ヨーダのが「ベストプラクティス」ではないことをします¹。 `null` がされないうり、ユニットテストのまたはバグレポートをるために `NullPointerException` するようにしてください。これにより、せぬ/ましくない `null` となったバグをつけてすることができます。

Yodaは、テストしているオブジェクトが`null`をすとしてされているAPIからているため、`null`がされるにのみするがあり`null`。そしていなく、それがすることができますので、テストをするあまり、かなりのいずれかのをするよりいかもしれない`null`あなたのコードをしているかにテストを。

1 - [Wikipedia](#)によると、「のコーディングは、ソフトウェアコミュニティがをかけてんだ、ソフトウェアのをさせるのにつのルールの設定です」。Yodaのをってもこれはしません。くの、コードがします。

オンラインでJavaのとし - NullとNullPointerExceptionをむ

<https://riptutorial.com/ja/java/topic/5680/javaのとし---nullとnullpointerexception>

34: Javaのとし - スレッドと

Examples

としwait/ notifyのった

メソッドobject.wait()、object.notify()およびobject.notifyAll()は、になでするためのものです。
。 <http://stackoverflow.com/documentation/java/5409/wait-notify#t=20160811161648303307>を

「われた」

なのいの1つは、にobject.wait()びすことobject.wait()

```
private final Object lock = new Object();

public void myConsumer() {
    synchronized (lock) {
        lock.wait();    // DON'T DO THIS!!
    }
    doSomething();
}
```

これがっているは、のスレッドがlock.notify()またはlock.notifyAll()をびすことにするが、lock.wait()というコンシューマスレッドののにのスレッドがそのびしをとっていないことをするものはないからです。

lock.notify()とlock.notifyAll()のスレッドがにをとっていないは、まったくもしません。こののmyConsumer()をびすスレッドは、をキャッチするにはすぎる、にハングします。

「な」のバグ

ロックをせずにオブジェクトにしてwait()またはnotify()をびすと、JVMはIllegalMonitorStateExceptionをスローしIllegalMonitorStateException。

```
public void myConsumer() {
    lock.wait();    // throws exception
    consume();
}

public void myProducer() {
    produce();
    lock.notify();    // throws exception
}
```

wait() / notify()のでは、システムをけるためにロックがされているがあります。ロックなしでwait()またはnotify()をびすことができた、これらのプリミティブのなユースケースがするのをつ

1かすぎる

`wait()`と`notify()`をけるもいは、それらをしないことです。ほとんどののは、`java.util.concurrent`パッケージでなレベルのオブジェクトキュー、バリア、セマフォなどをする
ことでできます。

とし - `java.lang.Thread`を

`Thread`クラスのjavadocは、`Thread`をしてする2つのをしています。

カスタムスレッドクラスをする

```
class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
        . . .
    }
}

PrimeThread p = new PrimeThread(143);
p.start();
```

Runnable

```
class PrimeRun implements Runnable {
    long minPrime;
    PrimeRun(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
        . . .
    }
}

PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```

ソース [java.lang.Thread javadoc](#)

カスタムスレッドクラスのアプローチはしますが、それにはいくつかのがあります

1. なスレッドプール、エグゼキュータ、またはForkJoinフレームワークをするコンテキストで
`PrimeThread`をするのはです。 `PrimeThread`に`Runnable`するのでではありませんが、`Runnable`と
してカスタム`Thread`クラスをすることはかに`PrimeThread`で、クラスののによってはではない
かもしれません。

2. のではいのがえます。あなたがしたたとえば、`PrimeThread.start()` にすることなく、`Thread.start()` あなたは、のスレッドでされていた「スレッド」でわるでしょう。

スレッドロジックを`Runnable`にくアプローチは、これらのをします。、クラスJava 1.1をして`Runnable`をすると、のはよりもでみやすくなります。

```
final long minPrime = ...
new Thread(new Runnable() {
    public void run() {
        // compute primes larger than minPrime
        . . .
    }
}).start();
```

ラムダJava 8では、のはよりエレガントになります。

```
final long minPrime = ...
new Thread(() -> {
    // compute primes larger than minPrime
    . . .
}).start();
```

とし。スレッドがすぎると、アプリケーションがくなります。

マルチスレッドにれていないのくは、スレッドをするとアプリケーションがにするとえています。、それよりもはるかにです。しかしにえることは、どのコンピュータでもにできるスレッドのにがあることです。

- コンピュータにはのコア またはハイパースレッド があります。
- するには、Javaスレッドをコアまたはハイパースレッドにスケジューリングするがあります。
- なJavaスレッドがなコア/ハイパースレッドよりもい、それらのうちのいくつかはするがあります。

これは、Javaスレッドをよりくするだけでは、アプリケーションをよりくにすることができないことをしています。しかし、にもすべきがあります。

- スレッドは、スレッドスタックにオフヒープメモリをとします。デフォルトのスレッドスタックサイズは512Kバイトまたは1Mバイトです。スレッドがい、メモリがきくなるがあります。
- アクティブなスレッドは、ヒープのいくつかのオブジェクトをします。これにより、なオブジェクトのワーキングセットがし、ガベージコレクションとメモリのにをえます。
- スレッドのりえのオーバーヘッドはではありません。これは、スレッドのスケジューリングをするためにOSカーネルにりえることをとする。
- スレッドとスレッドシグナリング`wait`、`notify`/`notifyAll`などのオーバーヘッドはなをつがあり

ます。

アプリケーションのみにして、これらのはにスレッドに「スイートスポット」があることをします。そのスレッドをすることで、パフォーマンスのがにえられ、パフォーマンスがするがあります。

しいタスクごとにアプリケーションをすると、しないのいなどがなにつながるがあります。

これにするよりいは、またはにサイズをできるスレッドプールをすることです。あまりにもくのがな、アプリケーションはをキューにれるがあります。 `ExecutorService` をすると、スレッドプールとタスクキューイングがされます。

とし。スレッドはです

の2つのマイクロベンチマークをえてみましょう。

のベンチマークはにスレッドを、、するだけです。スレッドの `Runnable` はしません。

```
public class ThreadTest {
    public static void main(String[] args) throws Exception {
        while (true) {
            long start = System.nanoTime();
            for (int i = 0; i < 100_000; i++) {
                Thread t = new Thread(new Runnable() {
                    public void run() {

                    }
                });
                t.start();
                t.join();
            }
            long end = System.nanoTime();
            System.out.println((end - start) / 100_000.0);
        }
    }
}
```

```
$ java ThreadTest
34627.91355
33596.66021
33661.19084
33699.44895
33603.097
33759.3928
33671.5719
33619.46809
33679.92508
33500.32862
33409.70188
33475.70541
33925.87848
33672.89529
^C
```

64ビットのJava 8 u101をしたLinuxをするなのPCでは、このベンチマークは、33.633.9マイクロのスレッドを、、するのにかかるをしています。

2のベンチマークはのものとですが、 `ExecutorService` をしてタスクをし、 `Future` をしてタスクのにランデブーをします。

```
import java.util.concurrent.*;

public class ExecutorTest {
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        while (true) {
            long start = System.nanoTime();
            for (int i = 0; i < 100_000; i++) {
                Future<?> future = exec.submit(new Runnable() {
                    public void run() {
                    }
                });
                future.get();
            }
            long end = System.nanoTime();
            System.out.println((end - start) / 100_000.0);
        }
    }
}
```

```
$ java ExecutorTest
6714.66053
5418.24901
5571.65213
5307.83651
5294.44132
5370.69978
5291.83493
5386.23932
5384.06842
5293.14126
5445.17405
5389.70685
^C
```

このように、は5.35.6マイクロです。

のはさまざまにしますが、これら2つののはです。しいスレッドをするよりも、スレッドプールをしてスレッドをリサイクルするがはるかにです。

としにはながです

このをえてみましょう。

```
public class ThreadTest implements Runnable {

    private boolean stop = false;

    public void run() {
        long counter = 0;
        while (!stop) {
            counter = counter + 1;
        }
    }
}
```

```

        System.out.println("Counted " + counter);
    }

    public static void main(String[] args) {
        ThreadTest tt = new ThreadTest();
        new Thread(tt).start();    // Create and start child thread
        Thread.sleep(1000);
        tt.stop = true;           // Tell child thread to stop.
    }
}

```

このプログラムのは、スレッドをし、1000ミリさせ、`stop`フラグをしてスレッドをさせることです。

それはしたとおりにしますか

たぶんそうだけとたぶんう。

`main`メソッドがしたときに、アプリケーションがするとはりません。のスレッドがされ、そのスレッドがデーモンスレッドとしてマークされていない、アプリケーションはメインスレッドがしたもききされます。このでは、アプリケーションがスレッドがするまでをすることをしています。これは、`tt.stop`が`true`されているにし`true`。

しかし、それはににはではありません。には、スレッドは、`true stop`をしたで`stop`し`true`。それはこるだろうかたぶんそうだけとたぶんう。

Javaでは、ソースコードのステートメントのどおりに、スレッドでわれたメモリのみみときみがそのスレッドからえることがされています。しかし、に、あるスレッドがきみをい、のスレッドでがみんだ、これはされません。をするには、きみとそれにくみみののにのこりのがするがあります。のでは、`stop`フラグへののためのチェーンはしないため、スレッドが`stop`を`true`することはされません。

へのなになるには、Javaメモリモデルにのトピックがです。

どのようにをするのですか

この、`stop`がにされるようにするには、2つのながあります。

1. `stop`を`volatile`するとします。すなわち、

```
private volatile boolean stop = false;
```

`volatile`の、JLSは、あるスレッドのきみとで2のスレッドのみみとのにこりうるがあることをします。

2. ミューテックスをして、のようにします。

```

public class ThreadTest implements Runnable {

    private boolean stop = false;

    public void run() {
        long counter = 0;
        while (true) {
            synchronize (this) {
                if (stop) {
                    break;
                }
            }
            counter = counter + 1;
        }
        System.out.println("Counted " + counter);
    }

    public static void main(String[] args) {
        ThreadTest tt = new ThreadTest();
        new Thread(tt).start();    // Create and start child thread
        Thread.sleep(1000);
        synchronize (tt) {
            tt.stop = true;        // Tell child thread to stop.
        }
    }
}

```

がすることをすることによって、JLSがこり、に 1つのスレッドでミューテックスをし、のスレッドでミューテックスをとのにながあることをします。

しかし、りてアトミックではありませんか

はい、そうです

しかし、そのは、のがすべてのスレッドににされることをするものではありません。ののなだけがそれをするでしょう。

なぜらはこれをやったのですか

めてJavaでマルチスレッドプログラミングをうプログラマは、メモリモデルがであるとじています。プログラムはにします。なぜなら、きみがにえるというのはのだからです。ではなぜJavaデザイナーがこのようにメモリモデルをするのですか

には、パフォーマンスといやすさのになありますプログラマーにとって。

のコンピューターアーキテクチャは、々のレジスタセットをするのプロセッサコアからなる。メインメモリは、すべてのプロセッサまたはプロセッサグループにアクセスできます。のコンピューターハードウェアのもう1つののは、レジスタへのアクセスが、メインメモリへのアクセスよりもはアクセスがであることである。コアのがえるにつれて、メインメモリへのみきがシステムのなパフォーマンスボトルネックになることはにわかります。

これは、プロセッサコアとメインメモリとの間に1つのレベルのメモリキャッシングをすることによってされます。コアは、キャッシュをしてメモリセルにアクセスします。、メインメモリのリードはキャッシュミスがしたにのみし、メインメモリライトはキャッシュラインをフラッシュするがあるにのみします。コアのメモリがキャッシュにまるアプリケーションの、コアのはもはやメモリの/によってされません。

しかし、これは、のコアがをみきしているときにしいをきこします。のバージョンは、1つのコアのキャッシュにかれることがあります。そのコアがキャッシュ・ラインをメイン・メモリにフラッシュしないり、のコアはキャッシュされたいバージョンのコピーをにしますが、のバージョンではいバージョンのがえます。しかし、キャッシュがメモリにフラッシュされるとメインメモリののをにるキャッシュきみのコアによるみしがあつたがします。

ハードウェアセットレベルでされるなは、キャッシュおよびキャッシュライトスルーのためのをし、それらをいつするかをするためにコンパイラにせることである。

Javaにる。メモリモデルは、Javaコンパイラがにはでないでキャッシュのとライトスルーをするがないようにされています。プログラマは、メモリのがであることをすために、なメカニズムプリミティブ`mutex`、`volatile`、よりいレベルのクラスなどをするをとしています。ながしない、Javaコンパイラはキャッシュまたはそれにしたものがないとにします。

これは、マルチスレッドアプリケーションにとってきなパフォーマンスのがありますが、しいマルチスレッドアプリケーションをすることはではないというがあります。プログラマは、がをしているのかをしなければなりません。

なぜこれをできないのですか

このようなをすることがしいはいくつかあります。

1. でしたように、メモリののをにできないというは、、コンパイルされたアプリケーションがメモリキャッシュをしくしないことになります。しかし、のように、メモリキャッシュはしばしばフラッシュされます。
2. ハードウェアプラットフォームをすると、メモリキャッシュのがわるがあります。これは、アプリケーションがしくしない、なるにつながります。
3. あなたはのをしているかもしれません。たとえば、トレースプリントをすると、、キャッシュフラッシュをきこすI/Oストリームのシーンのろでがします。したがって、をすると、アプリケーションのがなることがよくあります。
4. デバッガでアプリケーションをすると、JITコンパイラによってなるでコンパイルされます。ブレークポイントとシングルステップはこれをさせます。これらのエフェクトは、アプリケーションのをすることがよくあります。

これらのことは、なにするバグをすることがにである。

オンラインでJavaのとし - スレッドとをむ <https://riptutorial.com/ja/java/topic/5567/javaのとし--->

35: Javaのとし - パフォーマンスの

き

このトピックでは、Javaアプリケーションのパフォーマンスにするいくつかの「とし」つまり、のJavaプログラマがえているいについてします。

このトピックでは、な「マイクロ」Javaコーディングのについてします。ほとんどの、はいですが、それをけることはまだです。

Examples

とし - ログメッセージをするオーバーヘッド

TRACEとDEBUGログレベルは、にえられたコードのについていえることができます。ログレベルをこれらのレベルよりにすることがされますが、たは「オフ」になってもパフォーマンスにえなようにこれらのステートメントにするがあります。

このログステートメントをえてみましょう

```
// Processing a request of some kind, logging the parameters
LOG.debug("Request coming from " + myInetAddress.toString()
    + " parameters: " + Arrays.toString(veryLongParamArray));
```

ログレベルがINFOにされていても、debug()されたは、がされるたびにされます。これにより、いくつかのでにされます。

- StringのStringインスタンスがされます
- InetAddressはDNSルックアップをうもあります。
- veryLongParamArrayはにいかかもしれません - それからStringをするとメモリをし、がかります

ほとんどのロギング・フレームワークは、とオブジェクトをしてログ・メッセージをするをします。ログメッセージは、メッセージがにログにされたにのみされます。

```
// No toString() evaluation, no string concatenation if debug is disabled
LOG.debug("Request coming from {} parameters: {}", myInetAddress, parameters);
```

これは、[String.valueOfObject](#)をしてすべてのパラメータをにできるり、うまくします。ログメッセージcomputationがよりな、ログレベルをログにするにチェックすることができます。

```
if (LOG.isDebugEnabled()) {
    // Argument expression evaluated only when DEBUG is enabled
    LOG.debug("Request coming from {}, parameters: {}", myInetAddress,
```

```
Arrays.toString(veryLongParamArray);
}
```

ここでは、コストのかかる`Arrays.toString(Object[])`を`LOG.debug()`は、`DEBUG`がにになっているときにのみされます。

とし。ループののはされません

のコードをとしてえてみましょう。

```
public String joinWords(List<String> words) {
    String message = "";
    for (String word : words) {
        message = message + " " + word;
    }
    return message;
}
```

ながら、このコードは`words`リストがいとです。のはのとおりです。

```
message = message + " " + word;
```

ループで、このステートメントは、の`message`のすべてののコピーをむしい`message`をし、ながされます。これはなをたくさんし、くのコピーをいます。

`joinWords`をすると、さが M の N のがあるとすると、 ON のがされ、 OMN^2 がそのプロセスでコピーされることがわかります。 N^2 はにとなる。

このの1のされるアプローチは、のようにのわりに`StringBuilder`をすることです。

```
public String joinWords2(List<String> words) {
    StringBuilder message = new StringBuilder();
    for (String word : words) {
        message.append(" ").append(word);
    }
    return message.toString();
}
```

`joinWords2`のでは、`Builder`のをする`StringBuilder`バッキングを「`joinWords2`」するオーバーヘッドをすることがあります。しかし、されるしいオブジェクトのは $O\log N$ であり、コピーされるは OMN であることがわかります。は、の`toString()`びしでコピーされたをみます。

これをさらにするには、なの`StringBuilder`をすることがありますが、なさはわりません。

の`joinWords`メソッドにると、クリティカルなステートメントはなJavaコンパイラによってのうにされます。

```
StringBuilder tmp = new StringBuilder();
tmp.append(message).append(" ").append(word);
message = tmp.toString();
```

しかし、JavaコンパイラはjoinWords2のコードでいったように、StringBuilderループから「joinWords2ない」わけではありません。

- "ループのJavaの '+'はいですか

1 - Java 8では、こののをするために、Joinerクラスをできます。しかし、それはこのがにしているものではありません。

とし - 'new'をってプリミティブラッパーインスタンスをするのはです

Javaでは、Integer、Booleanなどのインスタンスをするためにnewをできますが、にはいえです。オートボクシングJava 5またはvalueOfメソッドをするがいです。

```
Integer i1 = new Integer(1);      // BAD
Integer i2 = 2;                   // BEST (autoboxing)
Integer i3 = Integer.valueOf(3);  // OK
```

new Integer(int)にするのは、JITコンパイラによってされていらないりしいオブジェクトをするということです。に、autoboxingまたはなvalueOfびしがされると、Javaランタイムは、のオブジェクトのキャッシュからIntegerオブジェクトをしようとします。にキャッシュ "ヒット"がするたびに、オブジェクトのがされます。これはまたヒープメモリをし、オブジェクトのれにするGCオーバーヘッドをします。

ノート

1. のJavaのでは、autoboxingはvalueOfをびすことでされ、Boolean、Byte、Short、Integer、Long、Characterキャッシュがあります。
2. のキャッシングは、Javaでされています。

とし - しいStringをびすことはです

をするためにnew String(String)をすることはでほとんどにです。

- Stringオブジェクトはなので、をぐためにコピーするはありません。
- いバージョンのJavaでは、Stringオブジェクトは、のStringオブジェクトとバッキングをできます。これらのバージョンでは、きののさをし、それをするによってメモリをリークすることがです。しかし、Java 7では、Stringバッキングはされません。

なメリットがない、new String(String)びすことはにです。

- コピーをするにはCPUがかかります。
- このコピーは、メモリのがえ、アプリケーションのメモがえたり、GCオーバーヘッドがしたりします。
- Stringオブジェクトをコピーすると、equals(Object)やhashCode() equals(Object)などのがくなるがあります。

とし - `System.gc()`のびしはです

`System.gc()`をびすのはほとんどのいえです。

`gc()`メソッドのjavadocは、をします。

「コール`gc`メソッドは、メソッドびしからがると、Javaマシンがするのをしたとき。Javaマシンは、らがのメモリーをすばやくなためにされていないオブジェクトをリサイクルことをしていますすべてのされたオブジェクトからのスペース。

これからりげることができるながいくつかあります

1. 「えてください」ではなく「」というをするというは、JVMがそのをすることがであることをします。デフォルトのJVMののリリースは、このにうことですが、JVMをするときに`-XX:+DisableExplicitGC`することでオーバーライドできます。
2. 「すべてのされたオブジェクトからをするためのの」というフレーズは、`gc`をびすと「な」ガベージコレクションがされることをします。

だから、なぜ`System.gc()`をいえとびますか

まず、なガベージコレクションをするのはです。なGCには、なすべてのオブジェクトをして「マークする」ことがまれます。つまり、ゴミではないすべてのオブジェクトです。するガベージがあまりないときにこれをトリガーすると、GCはほとんどがなく、くのをいます。

2に、なガベージコレクションは、されないオブジェクトの「」をするがある。ほぼじにじスレッドによってりてられたオブジェクトは、メモリーでにりてられるがあります。これはいい。にりてられたオブジェクトはしているがあります。すなわちいにする。アプリケーションでこれらのをしているは、さまざまなメモリーとページキャッシングのためにメモリーアクセスがになるがあります。ながら、なガベージコレクションはオブジェクトをさせるがあり、その、じることができたオブジェクトはさらにれてしまいます。

3に、なガベージコレクションをすると、コレクションがするまでアプリケーションがするがあります。これがこっている、あなたのアプリケーションはしません。

、のは、JVMがGCをいつするか、するコレクションのをさせることです。しない、JVMはスループットをするか、GCのをにするとタイプをします。

は "...ほとんどいつでもいえ..."とった。には、それがいいアイデアかもしれないいくつかのシナリオがあります

1. ガベージコレクションになコードえばファイナライザやい/らかい/ファントムをむもののテストをするは、`System.gc()`びすがあります。
2. いくつかのインタラクティブなアプリケーションでは、ガベージコレクションのがあるかどうかをユーザーがにしないのがするがあります。1つののは、「び」にながあるゲームである

。しいレベルをみむときなどにします。

とし、プリミティブラッパータイプのはです

の2つのコードをえてみましょう。

```
int a = 1000;
int b = a + 1;
```

そして

```
Integer a = 1000;
Integer b = a + 1;
```

どのバージョンがですか

2つのバージョンはほぼじですが、のバージョンは2つのバージョンよりもずっとです。

2のバージョンは、よりくのスペースをするのをしており、ボクシングとのアンボクシングについています。、2のバージョンはのコードにします。

```
Integer a = Integer.valueOf(1000);           // box 1000
Integer b = Integer.valueOf(a.intValue() + 1); // unbox 1000, add 1, box 1001
```

これを`int`をするのバージョンとすると、`Integer`がされているときに3つのなメソッドびしがあることはらかです。`valueOf`の、びしはそれぞれしい`Integer`オブジェクトをしてします。このなボクシングとアンボクシングのすべては、2のバージョンをのものよりも1くするがい。

それにえて、2のバージョンは、`valueOf`びしでヒープにオブジェクトをりてることです。スペースはプラットフォームですが、`Integer`オブジェクトの16バイトのにあるがあります。に、`int`バージョンは、`b`と`b`がローカルで`a`として`a`なヒープスペースをゼロにするがあります。

プリミティブがよりであるのきなは、ボックスされたが、それぞれのタイプがどのようにメモリにレイアウトされるかということです。

`int[]`と`Integer[]`をとると、`int[]`、`int`はメモリでしてされます。しかし、`Integer[]`、されるではなく、の`int`をむ`Integer`オブジェクトへのポインタです。

なレベルのにえて、これはをするときにキャッシュのになると、きなタンクになります。`int[]`の、CPUはメモリでしているため、のすべてのをにキャッシュにりむことができます。しかし、`Integer[]`の、にはのへのしかまれていないので、CPUはにしてのメモリフェッチをするがあります。

するに、プリミティブラッパーのは、CPUリソースとメモリリソースのです。それらをにすることはです。

とし、マップのキーをするとがくなることがある

のコードは、なよりもいです。

```
Map<String, String> map = new HashMap<>();
for (String key : map.keySet()) {
    String value = map.get(key);
    // Do something with key and value
}
```

これは、マップのキーにしてマップルックアップ `get()` メソッドがなためです。このルックアップはではないかもしれませんがHashMapでは、キーに`hashCode`をびし、データのしいバケットをし、によっては`equals`びすこともあります。なでは、これはなオーバーヘッドではないかもしれません。

これをするしいは、マップのエントリをりしすることです [コレクションのトピック](#)でしくしています。

とし、コレクションがであるかどうかをべるために`size`をうことはです。

Java Collections Frameworkは、すべての`Collection`オブジェクトにする2つのメソッドをします。

- `size()` は、`Collection`のエントリのをします。
- `isEmpty()` メソッドは、`Collection`がのにのみ`true`をします。

のメソッドをして、コレクションのをテストできます。えは

```
Collection<String> strings = new ArrayList<>();
boolean isEmpty_wrong = strings.size() == 0; // Avoid this
boolean isEmpty = strings.isEmpty(); // Best
```

これらのアプローチはじようにえますが、コレクションのにはサイズをしないものがあります。そのようなコレクションの、`size()`はびされるたびにサイズをするがあります。えは

- なリンクリストクラス `java.util.LinkedList`ではなくは、をえるためにリストをトラバースするがあります。
- `ConcurrentHashMap`クラスは、マップのすべての「セグメント」のエントリをするがあります。
- なコレクションのでは、をえるためにコレクションをメモリでするがあります。

に、`isEmpty()`メソッドは、コレクションになくとも1つのがあるかどうかをテストするだけです。これはをえることをとしない。

しながら、`size() == 0`にことあまりではない`isEmpty()`、それがにするためにえられない`isEmpty()`でに`size() == 0`。したがって、`isEmpty()`がされます。

とし、によるの

のマッチングはなツールJavaやそののコンテキストでですが、いくつかのがあります。これらのうちの1つは、がかなりになるがあります。

Pattern インスタンスと Matcher インスタンスをするがあります

のをえてみましょう。

```
/**
 * Test if all strings in a list consist of English letters and numbers.
 * @param strings the list to be checked
 * @return 'true' if an only if all strings satisfy the criteria
 * @throws NullPointerException if 'strings' is 'null' or a 'null' element.
 */
public boolean allAlphanumeric(List<String> strings) {
    for (String s : strings) {
        if (!s.matches("[A-Za-z0-9]*")) {
            return false;
        }
    }
    return true;
}
```

このコードはしいが、である。は、`matches(...)`びしににあります。フードのでは、`s.matches("[A-Za-z0-9]*")`はのようになります。

```
Pattern.matches(s, "[A-Za-z0-9]*")
```

これには

```
Pattern.compile("[A-Za-z0-9]*").matcher(s).matches()
```

`Pattern.compile("[A-Za-z0-9]*")`びしは、をし、し、エンジンでされるデータをする `Pattern` オブジェクトをします。これはなではありません。に、`s`をラップするために `Matcher` オブジェクトがされます。に `match()` をびしてのパターンマッチングをいます。

は、このがループのりしごとによりされることです。ソリューションは、のようにコードをすることです。

```
private static Pattern ALPHA_NUMERIC = Pattern.compile("[A-Za-z0-9]*");

public boolean allAlphanumeric(List<String> strings) {
    Matcher matcher = ALPHA_NUMERIC.matcher("");
    for (String s : strings) {
        matcher.reset(s);
        if (!matcher.matches()) {
            return false;
        }
    }
}
```

```
    return true;
}
```

Patternのjavadocはのようになることにしてください。

このクラスのインスタンスはであり、のスレッドでにできます。Matcherクラスのインスタンスは、このようなにはではありません。

findをうべきときは**match**をわないでください。

sに3のがまれているかどうかをテストしたいとします。あなたは、をむ々なでこれをします

```
if (s.matches("[0-9]{3}.*")) {
    System.out.println("matches");
}
```

または

```
if (Pattern.compile("[0-9]{3}").matcher(s).find()) {
    System.out.println("matches");
}
```

のがですが、がくなるもあります。それにして、のバージョンではをパターンとしようとしています。さらに、".*"は "greedy"パターンなので、パターンマッチャーはのまで"に"んで、マッチするまでバックトラックします。

に、2のバージョンはからへし、3のを付けるとすぐにをします。

よりなのをする

はなツールですが、あなたののツールではありません。くのタスクは、のでよりにできます。例えば

```
Pattern.compile("ABC").matcher(s).find()
```

じことをします

```
s.contains("ABC")
```

はずっとです。たとえをコンパイルするコストをすることができます。

しばしば、はよりです。たとえば、matches()メソッドがしたテストは、のallAlphanumericメソッドをのよういきすことができます。

```
public boolean matches(String s) {
    for (char c : s) {
```

```

        if ((c >= 'A' && c <= 'Z') ||
            (c >= 'a' && c <= 'z') ||
            (c >= '0' && c <= '9')) {
            return false;
        }
    }
    return true;
}

```

Matcher をするよりコードがくなりましたが、これもかなりになります。

なバックトラッキング

これは、のすべてのでになるがありますが、Pattern のとしてであるためここです。

このされたをえてみましょう

```

Pattern pat = Pattern.compile("(A+)+B");
System.out.println(pat.matcher("AAAAAAAAAAAAAAAAAAAAAAAAAB").matches());
System.out.println(pat.matcher("AAAAAAAAAAAAAAAAAAAAAAAAAC").matches());

```

のprintln はすぐにtrue。2のものはfalse。に。に、のコードをしてみると、C AにAをするた
びにが2になることがわかります。

これはなバックトラッキングのです。マッチングをするパターンマッチングエンジンは、パターン
がするのあるすべてのをなくしています。

(A+)+Bにするものをてみましょう。には、「1つのAとそれにくB」とわれていますが、にはそれぞ
れが1つのAでされる1つのグループがあります。したがって、たとえば

- 'AB'はにのみします 'AB'
- 「AAB」は、「AAB」または「AAB」という2つのにします。
- AAABまたはAAAB or '(A)(AA)B'またはAAABの4つのにする。
- 々

すれば、なのは 2^N であり、NはAのでA。

のははっきりとされていますが、このようなパフォーマンスつまり、きなKにして $O(2^N)$ または
 $O(N^K)$ をすパターンは、あまりされていないをするとにします。いくつかのながあります

- のりしパターンにりしパターンをれにしないでください。
- あまりにもくのりしパターンをしないでください。
- にじて、バックトラッキングのないリピートをしします。
- なにはをしないでください。わりになパーサーをしてください。

に、ユーザーまたはAPIクライアントがをつをできるにしてください。それは、またはな「サー
ビス」につながるがあります。

•

タグ、に<http://www.riptutorial.com/regex/topic/259/getting-started-with-regular-expressions/977/backtracking#t=201610010339131361163>および<http://www.riptutorial.com//トピック/259/をしてする/4527/すべきではないt=201610010339593564913>

- Jeff Atwoodの["Regex Performance"](#)
- Andreas Haufler、["でJavaをいかにすか"](#)

とし、`==`をうことができるようにするためのインターンはいえです

のプログラマーがこのアドバイスをと、

"`==`をしたのテストはですがされていないり"

らののは、`==`をうことができるように、をすることです。、`==`は`String.equals(...)`びすよりも`String.equals(...)`、そうではありません。

これは、さまざまなから、ったアプローチです。

まず、テストしているすべての`String`オブジェクトがインターンされていることをっていれば、`==`にうことができます。JLSは、ソースコードのリテラルがインターンされることをします。しかし、のJava SE APIのどれも、`String.intern(String)`とはに、`String.intern(String)`をすことをしていません。インターンされていない`String`オブジェクトのソースが1つしかつかからない、アプリケーションはできません。そのできないことは、それをするををよりにするがあるではなく、としてれる。

'intern'をするコスト

では、にされた`String`オブジェクトをむハッシュテーブルをメンテナンスすることによってします。ハッシュテーブルがストレージリークにならないように、らかのメカニズムがされます。ハッシュテーブルはネイティブコードでされていますが`HashMap`、`HashTable`などとはなります、`intern`コールはされるCPUとメモリにしてコストがかかります。

このコストは、`equals`わりに`==`をすることでられるとする`equals`ます。には、されたがのと「」されなければ、れません。

インターンシップがのあるいくつかのでは、じがもりされるアプリケーションのメモリフットプリントをらすがあり、これらののはくなります。

ガベージコレクションへの

のCPUとメモリのコストにえて、はガベージコレクタのパフォーマンスにします。

Java 7よりのバージョンのJavaの、されたは、まれにされる「PermGen」スペースにされます。PermGenをするがある、これははなガベージコレクションをトリガーします。PermGenスペースがにまると、のヒープスペースにきがあってもJVMがクラッシュします。

Java 7では、プールが "PermGen"からのヒープにされました。ただし、ハッシュテーブルはききされるデータになり、これによってのがになります。たとえされたオブジェクトがEdenにりてられたとしても、それらはされるにプロモートされるがもいでしょう。

したがって、すべてのにおいて、をインターンにすることは、のにべてのをばすことになります。これにより、JVMのにわたってガベージコレクションのオーバーヘッドがします。

2のは、ハッシュテーブルがのメモリをぐために、あるのいメカニズムをするがあることです。しかし、そのようなみはガベージコレクタのためのよりくのです。

これらのガベージコレクションのオーバーヘッドはするのがしいですが、することはほとんどありません。あなたがに`intern`をする、それらはながあります。

プールハッシュテーブルサイズ

[このソース](#)によれば、Java 6では、プールは、じバケットにハッシュするをうチェーンきのサイズのハッシュテーブルとしてされています。Java 6のリリースでは、ハッシュテーブルはハードされたのサイズをとっていました。チューニング・パラメータ `-XX:StringTableSize` がJava 6のとしてされました。その、Java 7のでは、プールのデフォルト・サイズが1009から60013にされました。

のは、あなたがするつもりならばということで`intern`あなたのコードに、ハッシュテーブルのサイズがであるところのJavaのバージョンをし、にそれをチューニングきさ、それをするをおめします。そうしないと、プールがきくなるにつれて、`intern`のパフォーマンスがするがあります。

なサービスのベクターとしてのインターン

のハッシュコードアルゴリズムはよくられています。のあるユーザーやアプリケーションからされたをすると、DoSDenial of Serviceのとしてされるがあります。のあるエージェントがするすべてののがじハッシュコードをとっているとすると、アンバランスなハッシュテーブルと`intern O(N)`パフォーマンスにつながります... N はしたのです。

DoSのがセキュリティをることであるや、DoSのをするは、このベクトルをすることができます。

とし、バッファされていないストリームにするさなみみ/きみはです

あるファイルのをファイルにコピーするには、のコードをしてください。

```
import java.io.*;

public class FileCopy {

    public static void main(String[] args) throws Exception {
        try (InputStream is = new FileInputStream(args[0]);
            OutputStream os = new FileOutputStream(args[1])) {
```



```

        int octet;
        while ((octet = is.read()) != -1) {
            os.write(octet);
        }
    }
}

```

このポイントにはしないので、のチェックやエラーなどをしてしています。

のコードをコンパイルしてなファイルをコピーする、にいいことがわかります。には、のOSファイルコピーユーティリティよりももくなるでしょう。

のパフォーマンスをここにしてください

のがいきなファイルのなは、バッファされていないバイトストリームにして1バイトのみりと1バイトのきみをしていることです。パフォーマンスをさせるなは、ストリームをバッファリングされたストリームでラップすることです。えは

```

import java.io.*;

public class FileCopy {

    public static void main(String[] args) throws Exception {
        try (InputStream is = new BufferedInputStream(
            new FileInputStream(args[0]));
            OutputStream os = new BufferedOutputStream(
            new FileOutputStream(args[1]))) {
            int octet;
            while ((octet = is.read()) != -1) {
                os.write(octet);
            }
        }
    }
}

```

これらのさなは、プラットフォームのさまざまなにじて、データのコピーをなくとももします。バッファリングされたストリームラッパーによって、データはよりきなチャンクでみきされます。インスタンスには、ともバイトとしてされたバッファがあります。

- で `is`、データはバッファにファイルからキロバイトをまれます。 `read()` がびされると、は、バッファからバイトをしします。バッファがになっているは、となるストリームからのみみみます。
- `os` のはしています。 `os.write(int)` びすと、バッファに1バイトがきまれます。データは、バッファがになったとき、または `os` がフラッシュされたとき、またはじられたときにのみ、ストリームにきまれます。

ベースのストリームはどうですか

Java I/Oは、バイナリデータとテキストデータのみきになるAPIをしています。

- `InputStream`と`OutputStream`はストリームベースのバイナリI/OのAPIです
- `Reader`と`Writer`はストリームベースのテキストI/OのAPIです。

テキストI/Oについては、`BufferedReader`と`BufferedWriter`ためのである`BufferedInputStream`と`BufferedOutputStream`。

なぜバッファリングされたストリームがこれをきくえるのでしょうか

バッファリングされたストリームがパフォーマンスをさせるのは、アプリケーションがオペレーティングシステムとするとすることです。

- JavaアプリケーションのJavaメソッド、またはJVMのネイティブ・ランタイム・ライブラリーのネイティブ・プロシージャータビしはです。には、いくつかのをり、パフォーマンスのはほとんどありません。
- に、オペレーティングシステムへのJVMランタイムコールはではありません。それらには「システムコール」とばれるものがまれます。システムコールのなパターンはのとおりです。

1. `syscall`をレジスタにれます。
2. `SYSENTER`トラップをします。
3. トラップハンドラはにりえ、メモリのマッピングをします。に、のシステムコールをするコードにディスパッチします。
4. `syscall`ハンドラは、ユーザプロセスがしてはならないメモリにアクセスするようにされていないことにして、をチェックします。
5. システムコールのがされます。 `read`システムの、これにはがまれます
 1. ファイルディスクリプタのでみむデータがあるかどうかをチェックする
 2. ファイルシステムハンドラをびしてなデータをディスクまたはされているからバッファキャッシュにフェッチし、
 3. バッファ・キャッシュからJVMのアドレスにデータをコピーする
 4. `thstream`ポインタファイルのをする
6. システムコールからるこれには、VMマッピングのとからのりえがです。

あなたがしているように、1つのシステムコールをするとものがされます。には、のメソッドびしよりもなくとも2い。おそらく3つ。

このことをえると、バッファリングされたストリームがきないをむのは、それらがシステムコールのをにらすからです。 `read()` びしごとに`syscall`をするわりに、バッファされたストリームはにじてのデータをバッファにみみます。バッファリングされたストリームのほとんどの`read()` びしは、にみまれた`byte`をチェックしてし`byte`。のがストリームの、およびストリームのにもてはまります。

バッファリングされたI/Oパフォーマンスは、みりサイズとディスクブロックのサイズ、ディスクなどのからていとえているもいます。アプリケーションは、ディスクをつはありませんが

、のではありません。

バッファリングされたストリームはにつのですか

にではない。バッファリングされたストリームは、あなたのアプリケーションがたくさんの "さな"みみやきみをう、いなくです。しかし、アプリケーションがきな `byte[]` または `char[]` とのみきをするがある、バッファされたストリームはあなたにのをえません。、さなパフォーマンスのペナルティさえあるかもしれません。

これは**Java**でファイルをコピーするもいですか

いいえ、そうではありません。JavaのストリームベースのAPIをしてファイルをコピーすると、データのメモリからメモリへのなコピーが1つになります。NIO `ByteBuffer` と `Channel` APIをするは、これをけることができます。のへのリンクをここにしてください。

オンラインでJavaのとし - パフォーマンスのをむ <https://riptutorial.com/ja/java/topic/5455/javaのとし---パフォーマンスの>

36: Javaのとし - の

き

いくつかのJavaプログラミングのは、しくコンパイルされているにもかかわらずなをするプログラムをすることがあります。このトピックのなは、にするなとしをリストし、そのようなとしをけるためのしいをすることです。

Examples

とし - をまたはす

このでは、にをするか、または「スカッシュ」します。あるいは、よりにええ、それをしてをしでするです。しかし、これをうをするに、まずをしつぶすことは、にそれらをするしいではないことをしておきます。

は、プログラムののにいくつかのなすなわち「な」イベントがしたことをするためにかによってスローされます。にいつもとはりませんが、はかがっていることをします。をスクラッシュするようにプログラムをコーディングすると、がのでびれるがあります。をさせるために、をスカッシュすると、オブジェクトのとそれにするスタックトレースがスローされます。それは、ののがであるかすることをにするがあります。

には、IDEのをしてのによってしたコンパイルエラーを「」すると、がにします。たとえば、ののようなコードがされます。

```
try {
    inputStream = new FileInputStream("someFile");
} catch (IOException e) {
    /* add exception handling code here */
}
```

らかに、プログラマはコンパイルエラーをすためにIDEのをけれましたが、はでした。の ""をうと、プログラムがでするがありますええ、 inputStreamがnullためにNullPointerExceptionしたNullPointerException。

いしましたが、にをしつぶすがあります。ののために、たちがセルフリーダーをせているにりみがあるのでとしたとしてください。このコメントは、にしつぶしたことをにえています。

```
try {
    selfie.show();
} catch (InterruptedException e) {
    // It doesn't matter if showing the selfie is interrupted.
}
```

なのをつけてこれをすのはどういふのか、にをしつぶしていることをするのです

```
try {
    selfie.show();
} catch (InterruptedException ignored) { }
```

が `ignored` され `ignored` いると、の IDE IntelliJ IDEA などにはのキャッチブロックにするがされません。

とし、エラーまたは **RuntimeException** の

のい Java プログラマーののえは、は「」または「」であり、これをするのはできるだけそれらをすべて 1 つにすることです。これはのようなコードにつながります

```
....
try {
    InputStream is = new FileInputStream(fileName);
    // process the input
} catch (Exception ex) {
    System.out.println("Could not open file " + fileName);
}
```

のコードにはながあります。 `catch` はにはプログラマがしているよりくのをキャッチしようとしています。アプリケーションののにあるバグのために、 `fileName` のが `null` であるとします。これにより、 `FileInputStream` コンストラクターは `NullPointerException` をスローします。ハンドラはこれをキャッチし、ユーザにします

```
Could not open file null
```

これはにたず、します。さらにいことに、しないチェックされているかチェックされていないをげたのは、それが "をする" コードだとします。これで、ユーザーは、ファイルをくときにしなかったのをくメッセージをけり、I/O にはまったくしなくなるがあります。

のは、プログラマが `Exception` ハンドラをコードしていることです。これはほとんどのいです。

- **Catching `Exception`** はチェックされたすべてのとチェックされていないもします。
- `RuntimeException` をキャッチすると、ほとんどチェックされていないがキャッチされます。
- **Catching `Error`** は、JVM のエラーをするチェックのをします。これらのエラーはにであり、まえるべきではありません。
- `catch Throwable` はすべてのをキャッチします。

あまりにもなをキャッチするは、ハンドラがそれらのすべてをにできないことです。 `Exception` などの、プログラマががされるかをすることはです。すなわち、をするか。

に、なはスローされたをすることです。たとえば、それらをキャッチしてそのでうことができます。

```
try {
    InputStream is = new FileInputStream(fileName);
    // process the input
} catch (FileNotFoundException ex) {
```

```
        System.out.println("Could not open file " + fileName);
    }
```

むで `throw` してすることもできます。

`Exception` キャッチするのがなはほとんどありません。にするのものは、のようなものです。

```
public static void main(String[] args) {
    try {
        // do stuff
    } catch (Exception ex) {
        System.err.println("Unfortunately an error has occurred. " +
                           "Please report this to X Y Z");
        // Write stacktrace to a log file.
        System.exit(1);
    }
}
```

ここではすべてののをにしたいので、 `Exception` または `Throwable` をキャッチするのはしいです。

1 - ポケモンともばれます。

とし - **Throwable**、`Exception`、`Error` または **RuntimeException** をげる

`Throwable`、`Exception`、`Error`、および `RuntimeException` をするのはいいですが、げることはさらにいいことです。

なは、アプリケーションでをすることがあるときに、トップレベルのがすると、さまざまなエラーをすることがになります。えは

```
try {
    InputStream is = new FileInputStream(someFile); // could throw IOException
    ...
    if (somethingBad) {
        throw new Exception(); // WRONG
    }
} catch (IOException ex) {
    System.err.println("cannot open ...");
} catch (Exception ex) {
    System.err.println("something bad happened"); // WRONG
}
```

は、`Exception` インスタンスをげたので、にそれをキャッチすることです。しかし、のでしたように、`Exception` キャッチするのはいいです。このでは、`somethingBad` が `true` にスローされる `Exception` 「される」ケースと、`NullPointerException` などのチェックののをにキャッチするしないケースをすることがになります。

のがすることがされているは、のがします。

- たちは、トップレベルをげて、それを/うさまざまなのすべてをえなければなりません。

- `Exception`と`Throwable`、をさせたいは、これらのをメソッドの`throws`にするもあります。これはのようになっています。

つまり、これらのをスローしないでください。こった"な"をよりにする、よりなをけてください。なは、カスタムクラスをしてします。

メソッドの **"throws"**で`Throwable`または`Exception`をするのはあります。

メソッドの`throws`で`throws`れたのいリストを`Exception`または`Throwable`にきえることはです。これはいえです

1. これは、びしに`Exception`をまたはさせ`Exception`。
2. たちはもはや、するがあるのチェックについてたちにえるためにコンパイラにすることはできません。
3. `Exception`しくするのはしいです。のがどのようにされるのかをるのはしく、がまえることができないのかわからない、がであるかどうかをすることはしいです。
4. `Throwable`はさらにしくなります。なぜなら、してできないのあるにしなければならないからです。

このアドバイスは、のパターンをけるべきことをします。えは

```
try {
    doSomething();
} catch (Exception ex) {
    report(ex);
    throw ex;
}
```

では、すべてのをにせずにすべてのをログにしようとしています。ながら、Java 7のには、`throw ex;`ステートメントによって、コンパイラーは`Exception`がスローされるとえるようになりました。これは、みメソッドを`throws Exception`としてするようする`throws Exception`。Java 7では、コンパイラーは、スローされるセットがよりさくすることをしています。

とし - `InterruptedException`をキャッチする

にのとしでされているように、

```
try {
    // Some code
} catch (Exception) {
    // Some error handling
}
```

さまざまながたくさんあります。しかし、1つの`perticular`は、マルチスレッドアプリケーションをくときによりみシステムをするので、デッドロックにつながるがあるということです。

スレッドをするは、、さまざまなですがあります。

```
Thread t = new Thread(new Runnable() {
    public void run() {
        while (true) {
            //Do something indefinitely
        }
    }
});

t.start();

//Do something else

// The thread should be canceled if it is still active.
// A Better way to solve this is with a shared variable that is tested
// regularly by the thread for a clean exit, but for this example we try to
// forcibly interrupt this thread.
if (t.isAlive()) {
    t.interrupt();
    t.join();
}

//Continue with program
```

`t.interrupt()` はスレッドをシャットダウンするよりも、そのスレッドで `InterruptedException` をさせます。しかし、スレッドがににするにいくつかのリソースをクリーンアップするがあるはどうしますかこのため、`InterruptedException` をしてクリーンアップをうことができます。

```
Thread t = new Thread(new Runnable() {
    public void run() {
        try {
            while (true) {
                //Do something indefinitely
            }
        } catch (InterruptedException ex) {
            //Do some quick cleanup

            // In this case a simple return would do.
            // But if you are not 100% sure that the thread ends after
            // catching the InterruptedException you will need to raise another
            // one for the layers surrounding this code.
            Thread.currentThread().interrupt();
        }
    }
});
```

しかし、コードに `catch-all` がある、`InterruptedException` もにキャッチされ、はされません。スレッドが `t.join()` するのをにつため、これはデッドロックになるがあります。

```
Thread t = new Thread(new Runnable() {
    public void run() {
        try {
            while (true) {
                try {
                    //Do something indefinitely
```



```

        }
        catch (Exception ex) {
            ex.printStackTrace();
        }
    }
} catch (InterruptedException ex) {
    // Dead code as the interrupt exception was already caught in
    // the inner try-catch
    Thread.currentThread().interrupt();
}
}
}

```

したがって、々にをすることがありますが、**catch-all**をすることをくめているは、なくとも**InterruptedException**をにしてください。

```

Thread t = new Thread(new Runnable() {
    public void run() {
        try {
            while (true) {
                try {
                    //Do something indefinitely
                } catch (InterruptedException ex) {
                    throw ex; //Send it up in the chain
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
            }
        } catch (InterruptedException ex) {
            // Some quick cleanup code

            Thread.currentThread().interrupt();
        }
    }
}
}

```

とし、のフローコントロールにするの

Javaののには、

「は、なにのみすべきです。

<http://programmers.stackexchange.com/questions/184654>

これは、やをとってのフローをするのはいえですJavaの。たとえば、**null**ののあるパラメータをう2つのをしてください。

```

public String truncateWordOrNull(String word, int maxLength) {
    if (word == null) {
        return "";
    } else {
        return word.substring(0, Math.min(word.length(), maxLength));
    }
}

```

```
public String truncateWordOrNull(String word, int maxLength) {
    try {
        return word.substring(0, Math.min(word.length(), maxLength));
    } catch (NullPointerException ex) {
        return "";
    }
}
```

ここでは、`word`が`null`であるかのように、`word`が`null`をに`いnull`。2つのバージョンが`うnull`、の... `else`とか...キャッチをみるしてどちらか。どんなバージョンがれているかをどのようにするべきですか

のはみやすさです。はにすることはしいですが、ほとんどのプログラマはのバージョンのながかりやすいことにします。に、2のをにするには、`Math.min`または`String.substring`メソッドで`NullPointerException`をスローすることができないことをするがあります。

2のはである。Java 8よりのJavaのリリースでは、2のバージョンはのバージョンよりもににいます。に、オブジェクトのには、スタックトレースがなにえて、スタックフレームをキャプチャしてするがあります。

、をすることは、「な」イベントにするためにきコードをするよりもみやすく、で、にはよりなくのがあります。、「でない」イベントにそれらをするがあるまれながあります。にするである。の、オブジェクトをするオーバーヘッドをらすをするがあります。

とし - またはなみね

プログラマがうことができるよりなことの1つは、コードをして`printStackTrace()`びしをさせることです。

は、`printStackTrace()`がスタックトレースをにきむことです。

- Javaプログラマではないエンドユーザをとしたアプリケーションの、スタックトレースはであり、のにはです。
- サーバーのアプリケーションでは、もをないがあります。

よりいいアイデアは、`printStackTrace`びさないか、それをびすは、スタックトレースがエンドユーザのコンソールではなくログファイルまたはエラーファイルにきまれるようにします。

これをう1つの、ロギングフレームワークをし、オブジェクトをログイベントのパラメータとしてすことです。しかし、をログにすることさえも、ながるとになるがあります。のをしてください。

```
public void method1() throws SomeException {
    try {
        method2();
        // Do something
    } catch (SomeException ex) {
        Logger.getLogger().warn("Something bad in method1", ex);
    }
}
```

```

        throw ex;
    }
}

public void method2() throws SomeException {
    try {
        // Do something else
    } catch (SomeException ex) {
        Logger.getLogger().warn("Something bad in method2", ex);
        throw ex;
    }
}

```

method2でがスローされた、じにするログファイルにじスタックトレースの2つのコピーがされるがあります。

するに、をするか、それをさらにスローしますのでラップされるがあります。をしないでください。

とし - `Throwable`をサブクラスする

Throwableは、2つのなサブクラスExceptionとErrorます。Throwableするしいクラスをすることはですが、くのアプリケーションでExceptionとErrorだけがするとされるため、これはおめできません。

には、Throwableをサブクラスすることにはにはメリットがありません。としてられるクラスにはにはチェックです。わりにExceptionをサブクラスすると、じになりますが、よりにあなたのをえます。

オンラインでJavaのとし - のをむ <https://riptutorial.com/ja/java/topic/5381/javaのとし---の>

37: Javaのとし - の

き

いくつかのJavaプログラミングのは、しくコンパイルされているにもかかわらずなをするプログラムをすることがあります。このトピックのなは、なとしをそのとともにげ、そのようなをけるしいをすることです。

このトピックでは、エラーがしやすいJava、またはのでしないJavaののについてします。

Examples

- メソッドのをする

なJavaでも、Javaには3つのしかないといえるがあります。はに4つです パッケージプライベートデフォルトのレベルはしばしばられます。

どんなでするかをうがあります。アプリケーションのパブリックメソッドは、アプリケーションのAPIです。これは、になライブラリをしているは、できるだけさく、コンパクトにするがあります **SOLID**もしてください。に、すべてのメソッドのをし、にじてされたプライベートアクセスのみをすることがです。

privateとしてするメソッドをすると、クラスのがされます。

これには、クラスのパブリックメソッドを**テストでテスト**することしかできません。は、パブリックメソッドのみをテストできます。これらのメソッドにしてテストをできるようにするために、プライベートメソッドのをめることはいいです。よりなをつメソッドをびすパブリックメソッドをテストすることは、APIをテストするのになはずです。テストをにするためだけに、メソッドをしてAPIをするべきではありません。

とし - 「スイッチ」ケースで「ブレーク」がない

これらのJavaのはになことがあり、プロダクションでするまでのまになることがあります。switchステートメントでのなはしばしばにちます。しかし、そのようながましくないときに ""キーワードがないと、なにつながるがあります。のコードで "case 0"に "break"をれるのをれた、プログラムは "Zero"と "One"をきます。ここでのフローは "switch"ステートメントをするでしょう。それは ""にする。えは

```
public static void switchCasePrimer() {
    int caseIndex = 0;
    switch (caseIndex) {
        case 0:
            System.out.println("Zero");
        case 1:
```

```

        System.out.println("One");
        break;
    case 2:
        System.out.println("Two");
        break;
    default:
        System.out.println("Default");
    }
}

```

ほとんどの、よりされたソリューションは、インターフェイスをし、のをコードをのをえたにすることです。

switchがやむをないは、「される」フォールススルーがしたにすることをおめします。そうすれば、のにあなたがけていることをしていることをすことができます。これはされるです。

```

switch(caseIndex) {
    [...]
    case 2:
        System.out.println("Two");
        // fallthrough
    default:
        System.out.println("Default");
}

```

とし - ったセミコロンとけ

これは、なくともJavaにとってはにをくいです。これをくわりに

```

if (feeling == HAPPY)
    System.out.println("Smile");
else
    System.out.println("Frown");

```

らはってこれをいています

```

if (feeling == HAPPY);
    System.out.println("Smile");
else
    System.out.println("Frown");

```

Javaコンパイラがelseがっているとえとします。Javaコンパイラは、をのようになります。

```

if (feeling == HAPPY)
    /*empty statement*/ ;
System.out.println("Smile"); // This is unconditional
else                        // This is misplaced. A statement cannot
                           // start with 'else'
System.out.println("Frown");

```

のケースでは、コンパイルエラーはしませんが、コードはプログラマがするものをしません。え

```
for (int i = 0; i < 5; i++);  
    System.out.println("Hello");
```

だけ"こんにちは"をします。もう、のセミコロンは、`for`ループのがであることをします。これは、の`println`びしがであることをします。

のバリエーション

```
for (int i = 0; i < 5; i++);  
    System.out.println("The number is " + i);
```

これにより、`i`にして「シンボルをつけることができません」というエラーがされます。セミコロンのは、`println`びしがそのスコープので`i`をしようとしていることをします。

これらのでは、ながあります。のセミコロンをするだけです。しかし、これらのからきされるいくつかのいがあります

1. Javaのセミコロンは "なノイズ"ではありません。セミコロンのは、プログラムのをえることができます。すべてのにそれらをするだけではありません。
2. コードのインデントをししないでください。Javaでは、のなはコンパイラーによってされます。
3. をします。すべてのIDEとくのなテキストエディタは、Javaコードをしくインデントするをしています。
4. これがもなです。のJavaスタイルのガイドラインにい、"then"ステートメントと"else"ステートメントとループのbodyステートメントのにカッコをれてください。{ をしてはいけません。

プログラマーがスタイル・ルールにつた`if`、つたセミコロンをむ`if`はのようになります。

```
if (feeling == HAPPY); {  
    System.out.println("Smile");  
} else {  
    System.out.println("Frown");  
}
```

かなにはにえます。そのコードをインデントしたは、のようになります。

```
if (feeling == HAPPY); {  
    System.out.println("Smile");  
} else {  
    System.out.println("Frown");  
}
```

にもっているはずです。

とし、をして「ぶらがっている」と「がいている」

Oracle Javaスタイルガイドのバージョンでは、`if`ステートメントの "then"ステートメントと "else"ステートメントをに "かっこ"または ""でむがあります。さまざまなループのにものがされます。

```
if (a) {           // <- open brace
    doSomething();
    doSomeMore();
}                 // <- close brace
```

これはJavaのではありません。、`if`の "then"がのである`if`、をすることはです

```
if (a)
    doSomething();
```

あるいは

```
if (a) doSomething();
```

しかし、Javaスタイルのルールをしてカッコをいってしまうがあります。には、ったインデントのコードがっているというリスクをにさせます。

「ぶらがっている」

ののコードを、なしできしてみましよう。

```
if (a)
    doSomething();
    doSomeMore();
```

このコードは、へのびしとっているようだ `doSomething`と`doSomeMore`のにのみしますです `a true`。、コードはってインデントされています。 `doSomeMore()`びしが`if`ステートメントにくのステートメントであるJava。しいげはのとおりです。

```
if (a)
    doSomething();
doSomeMore();
```

"ぶらがっている"

ミックスに`else`をすると、2のがします。のがないをえてみましよう。

```
if (a)
    if (b)
        doX();
    else if (c)
        doY();
else
    doZ();
```

のコードは、`a`が`false`ときに`doZ`がびされるとい`false`。には、インデントがっています。コードのインデントはのとおりです。

```
if (a)
  if (b)
    doX();
  else if (c)
    doY();
  else
    doZ();
```

コードがJavaスタイルのルールによってされている、はのようになります。

```
if (a) {
  if (b) {
    doX();
  } else if (c) {
    doY();
  } else {
    doZ();
  }
}
```

なぜそれがいのかをするために、ってコードをってインデントしたとします。あなたはこのようなものでわるかもしれません

<pre>if (a) { if (b) { doX(); } else if (c) { doY(); } else { doZ(); } }</pre>	<pre>if (a) { if (b) { doX(); } else if (c) { doY(); } else { doZ(); } }</pre>

しかし、いずれのも、ってインデントされたコードは、なJavaプログラマーのにはっています。

とし - オーバーライドではなくオーバーロード

のをえてみましょう。

```
public final class Person {
  private final String firstName;
  private final String lastName;

  public Person(String firstName, String lastName) {
    this.firstName = (firstName == null) ? "" : firstName;
    this.lastName = (lastName == null) ? "" : lastName;
  }

  public boolean equals(String other) {
    if (!(other instanceof Person)) {
      return false;
    }
  }
}
```



```

    }
    Person p = (Person) other;
    return firstName.equals(p.firstName) &&
           lastName.equals(p.lastName);
}

public int hashCode() {
    return firstName.hashCode() + 31 * lastName.hashCode();
}
}

```

このコードはどおりにしません。は、 `Person` `equals` メソッドと `hashCode` メソッドが `Object` されたメソッドをオーバーライドしないことです。

- `equals` メソッドのシグネチャがっています。 `equals(Object)` **not** `equals(String)` としてするがあります。
- `hashCode` メソッドのがっています。これは `hashCode()` なければなりません `hashCode()` の **C** にしてください。

これらのいは、なオーバーロードをしたことをし、 `Person` がコンテキストでされているはされません。

しかし、これにするながあります **Java 5**。 `@Override` あなたがオーバーライドするあなたのをしたときにを

Java SE 5

```

public final class Person {
    ...

    @Override
    public boolean equals(String other) {
        ....
    }

    @Override
    public hashCode() {
        ....
    }
}

```

`@Override` アノテーションをメソッドにすると、コンパイラはメソッドがスーパークラスまたはインタフェースでされたメソッドをオーバーライドまたはするかどうかをチェックします。のでは、コンパイラは2つのコンパイルエラーをします。コンパイルエラーはたちにいをするのになはずです。

とし - **8** リテラル

のコードスニペットをえてみましょう。

```

// Print the sum of the numbers 1 to 10
int count = 0;

```

```
for (int i = 1; i < 010; i++) {    // Mistake here ....
    count = count + i;
}
System.out.println("The sum of 1 to 10 is " + count);
```

Javaのは、のプログラムがったえをしていることにくかもしれません。には1から8ののをします。

そのは、ゼロ '0'でまるリテラルは、Javaコンパイラによってされるように10リテラルではなく8リテラルとしてされるためです。って、010は810であり、これは10で8である。

とし。クラスとじのクラスをする

には、Javaをめてするプログラマーは、くわれているクラスとじのクラスをするのをえます。え

```
package com.example;

/**
 * My string utilities
 */
public class String {
    ....
}
```

そして、らはせぬエラーがするをります。え

```
package com.example;

public class Test {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

のクラスをコンパイルしてしようとすると、エラーがします

```
$ javac com/example/*.java
$ java com.example.Test
Error: Main method not found in class test.Test, please define the main method as:
    public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```

かがTestクラスのコードをて、mainのをて、そのをて、javaコマンドがをにしているのかにいます。しかし、には、javaコマンドはをえています。

TestとじパッケージにStringバージョンをすると、このバージョンがjava.lang.Stringインポートよりもされjava.lang.String。したがって、Test.mainメソッドのシグネチャはには

```
void main(com.example.String[] args)
```

のわりに

```
void main(java.lang.String[] args)
```

java コマンドはentrypointメソッドとしてそれをしません。

レッスン `java.lang` のクラスとじをつクラス、またはJava SEライブラリのものよくされるクラスをしないでください。あなたがそうするならば、あなたはあらゆるのあからさまなエラーのためにをいています。

とし `'=='` をってブールをテストする

しいJavaプログラマは、のようなコードをくことがあります。

```
public void check(boolean ok) {
    if (ok == true) {           // Note 'ok == true'
        System.out.println("It is OK");
    }
}
```

なプログラマーは、それがで、それをのようきしたいとうでしょう

```
public void check(boolean ok) {
    if (ok) {
        System.out.println("It is OK");
    }
}
```

しかし、なさよりも `ok == true` がっています。このバリエーションをえてみましょう。

```
public void check(boolean ok) {
    if (ok = true) {           // Oooops!
        System.out.println("It is OK");
    }
}
```

ここでプログラマーは `== as = ...` をってしましたが、コードにはなバグがあります。 `x = true` に `x` `true` をしてから `true` し `true` 。 いえれば、 `check` メソッドは、パラメータがであっても "それはOK" をするようになりました。

ここでののは、 `== false` と `== true` をするからけすこと `== true` 。 であることにえて、コーディングはエラーをこしやすくなります。

としをけるための `ok == true` わりに、 **Yoda** をすることもできます。すなわち、 `true == ok` ように、リテラルをのにく。これはうまくいくが、ほとんどのプログラマーはヨードのがにえることにおそらくするだろう。かに `ok` または `!ok` はよりです。

とし、ワイルドカードのインポートによってコードがれやすくなる

のなをえてみましょう。

```
import com.example.somelib.*;
import com.acme.otherlib.*;

public class Test {
    private Context x = new Context();    // from com.example.somelib
    ...
}
```

に `somelib` バージョン 1.0 と `somelib` バージョン 1.0 にしてコードをしたときを `otherlib`。その、あるでをしいバージョンにアップグレードし、`otherlib` バージョン 2.0 をすることにします。また、1.0 と 2.0 ので `otherlib` えたの 1 つが `Context` クラスをすることだったとします。

`Test` をコンパイルすると、`Context` があいまいなインポートであることをすコンパイルエラーがします。

あなたがコードベースにしているなら、これはおそらくほんのしのです。そうでなければ、あなたはこのにりむためにいくつかのをいます。

ここでの、ワイルドカードのインポートです。では、ワイルドカードをすると、クラスをくすることができます。

- コードベースの、Java ライブラリ、またはサードパーティのライブラリへののは、コンパイルエラーにつながるがあります。
- みやすさにがあります。IDE をしているをき、ワイルドカードのインポートのどれがきクラスをしているかをするのはいいことがあります。

は、するがあるコードでワイルドカードのインポートをすることはいえです。のワイルドカードではないインポートは、IDE をしているにはあまりするものではなく、そのはがあります。

としやユーザーのに **'assert'** をう

StackOverflow では、メソッドにされたをするために `assert` を `assert` がであるか、ユーザーによってされるさえかどうかというがあります。

なえはでないということです。

よりいいのとおりです。

- カスタムコードをして `IllegalArgumentException` をスローします。
- Google Guava ライブラリでな `Preconditions` メソッドの。
- Apache Commons Lang3 ライブラリでな `Validate` メソッドの。

これは、[Java JLS 14.10](#)、[for Java 8](#) がこのについてアドバイスしているものです。

、アサーションのチェックは、プログラムのおよびテストにになり、をにしてパフォーマンスをさせます。

アサーションがになるがあるため、プログラムはアサーションにまれるがされることをしてはいけません。したがって、これらのブールはにがないはずです。このようなブールをしても、がしたにされるにはしません。アサーションにまれるブールがをつことはではありませんが、アサーションがかかによってプログラムのがなるがあるため、はです。

これにらして、パブリックメソッドのチェックにアサーションをすべきではありません。チェックは、メソッドのコントラクトのであり、アサーションがであるかであるかにわらず、このをするがあります。

チェックにアサーションをするの2のは、つたがな `IllegalArgumentException`、`ArrayIndexOutOfBoundsException`、または `NullPointerException` になることです。アサーションのはなをスローしません。りしますが、パブリックメソッドのチェックにアサーションをすることはではありませんが、はです。 `AssertionError` はしてまえないことをしていますが、そうすることができます。したがって、tryステートメントのルールは、のthrowステートメントのいとに、tryブロックにれるアサーションをするがあります。

プリミティブへのヌルオブジェクトのアンボクシングのとし

```
public class FooBar {
    public static void main(String[] args) {

        // example:
        Boolean ignore = null;
        if (ignore == false) {
            System.out.println("Do not ignore!");
        }
    }
}
```

ここでのとしは、`null`が`false`とされる`false`です。プリミティブ`boolean`と`Boolean` `boolean`するので、Javaは`Boolean` Objectをプリミティブ`Boolean`にアンボックスしようとします。これはのができています。ただし、そのは`null`であるため、`NullPointerException`がスローされます。

Javaでは、プリミティブと`null`をできないため、に`NullPointerException`します。`false == null` プリミティブのをえてみましょう。これは、コンパイルに、`incomparable types: int and <null>`。

オンラインでJavaのとし - のをむ <https://riptutorial.com/ja/java/topic/5382/javaのとし---の>

38: Javaパフォーマンスチューニング

Examples

インターネットには、Javaプログラムのパフォーマンスのヒントがされています。1のヒントはです。つまり、

- パフォーマンスのとボトルネックをします。
- ツールとテストツールをする。
- いといをる。

しいシステムやモジュールについて、のポイントはでうがあります。レガシーコードについてと、ツールとテストツールがします。JVMのパフォーマンスをするためのもなツールは、JVisualVMです。これはJDKにまれています。

は、となについて、など、このページなどにされますもちろんのヒント、のにある[この](#)。

ストリングのをらす

Javaでは、とされなくのStringインスタンスをするのはすぎます。それで、プログラムでGCがしくなっているストリングがたくさんあるがあります。

Stringインスタンスをするはいくつかあります。

```
myString += "foo";
```

ループやでは、いことに、

```
for (int i = 0; i < N; i++) {  
    myString += "foo" + i;  
}
```

は、+がしいStringをすることです、しいコンパイラはいくつかのケースをするので。

StringBuilderまたはStringBufferをしてをうことができます。

```
StringBuffer sb = new StringBuffer(myString);  
for (int i = 0; i < N; i++) {  
    sb.append("foo").append(i);  
}  
myString = sb.toString();
```

いStringたとえばSQLをにするは、StringAPIをします。

すべきの

- replace、substringなどのをらす
-

にアクセスするコードでは、`String.toArray()` けてください。

- フィルタリングすることとしたログえば、ログレベルのためはしないでくださいログレベルはにチェックするがあります。
- にじて、 [この](#)ようなライブラリをしてください。
- かのスレッドでされる、`StringBuilder`がれています。

Javaのパフォーマンスチューニングにするベースのアプローチ

ドナルド・クヌスはこれをのようになっています。

「プログラマはについてえるのなをにし、またはにせ、そのプログラムののいの、およびデバッグ、メンテナンスのでこれらのみは、にいのをえることがえられている。々はさなをれなければならない、についています97の のはすべてののですが、な3でをすべきではありません。

ソース

sageにするアドバイスをにいて、ここではプログラムをするためにされるです

1. まず、シンプルさとさにをてて、プログラムやライブラリをし、コーディングします。まず、パフォーマンスになをわないでください。
2. それをにし、コードベースののテストをすることがです。
3. アプリケーションレベルのパフォーマンスベンチマークをする。ベンチマークは、アプリケーションのパフォーマンスのなをカバーし、アプリケーションをでどのようにするかのなのをするがあります。
4. パフォーマンスをします。
5. されたパフォーマンスと、アプリケーションがどれくらいになるがあるかにするとをしてください。「できるだけ」などのな、な、またはなをける
6. をたすは、してください。あなたはがわった。それのはおそらくのです。
7. アプリケーションがパフォーマンスベンチマークをしているにプロファイリングします。
8. プロファイリングのをべ、のされていない「パフォーマンスのホットスポット」をします。すなわち、アプリケーションがもをやしているとわれるコードのセクション。
9. ホットスポットコードセクションをして、なぜボトルネックであるのかをし、するをえてください。
10. コード、テスト、デバッグとしてします。
11. ベンチマークをして、コードのによってパフォーマンスがしたかどうかをします。

- はいのは、4にります。

- いいえのは、をしてgにります。がないは、のホットスポットをしてしてください。

には、アプリケーションのがいか、なホットスポットのすべてをしたになります。この、このアプローチをめるがあります。コードのがのの1をしている、50のでさえ、アプリケーションをに0.5くするだけです。

らかに、ホットスポットのがあるのであります。そのにすれば、よりなアプローチをるがあります。えは

- コアアルゴリズムのアルゴリズムのさをてください。
- アプリケーションがガベージコレクションにくのをやしているは、オブジェクトのをさせるをします。
- アプリケーションのながCPUでシングルスレッドのは、のをします。
- アプリケーションがすでにマルチスレッドのは、のボトルネックをします。

しかし、なり、あなたのをするためにではなく、ツールとってください。

オンラインでJavaパフォーマンスチューニングをむ <https://riptutorial.com/ja/java/topic/4160/javaパフォーマンスチューニング>

39: Java プラグインシステムの

IDEやビルドシステムをしているは、このプロジェクトをするがはるかにです。メインのアプリケーションモジュールをしてからAPIモジュールをし、プラグインモジュールをして、APIモジュールまたはそのにさせます。に、プロジェクトのをどこにするかをします。コンパイルされたプラグインのjarファイルを 'plugins'ディレクトリにまっすぐことができ、でのをけることができます。

Examples

URLClassLoaderの

Javaアプリケーションのプラグインシステムをするはいくつかあります。 *URLClassLoader*をするのがものの1つです。のでは、JavaFXコードをします。

なアプリケーションのモジュールがあるとします。このモジュールは 'plugins'フォルダからJarsのでプラグインをみむことになっています。コード

```
package main;

public class MainApplication extends Application
{
    @Override
    public void start(Stage primaryStage) throws Exception
    {
        File pluginDirectory=new File("plugins"); //arbitrary directory
        if(!pluginDirectory.exists())pluginDirectory.mkdir();
        VBox loadedPlugins=new VBox(6); //a container to show the visual info later
        Rectangle2D screenbounds=Screen.getPrimary().getVisualBounds();
        Scene scene=new
        Scene(loadedPlugins,screenbounds.getWidth()/2,screenbounds.getHeight()/2);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    public static void main(String[] a)
    {
        launch(a);
    }
}
```

に、プラグインをすインターフェースをします。

```
package main;

public interface Plugin
{
    default void initialize()
    {
        System.out.println("Initialized "+this.getClass().getName());
    }
    default String name(){return getClass().getSimpleName();}
}
```

```
}
```

このインタフェースをするクラスをロードしたいので、に '.jar'をつファイルをフィルタリングするがあります

```
File[] files=pluginDirectory.listFiles((dir, name) -> name.endsWith(".jar"));
```

ファイルがあるは、URLとクラスのコレクションをするがあります。

```
if(files!=null && files.length>0)
{
    ArrayList<String> classes=new ArrayList<>();
    ArrayList<URL> urls=new ArrayList<>(files.length);
    for(File file:files)
    {
        JarFile jar=new JarFile(file);
        jar.stream().forEach(jarEntry -> {
            if(jarEntry.getName().endsWith(".class"))
            {
                classes.add(jarEntry.getName());
            }
        });
        URL url=file.toURI().toURL();
        urls.add(url);
    }
}
```

ロードされたプラグインをする *MainApplication* にな *HashSet* をしましょう

```
static HashSet<Plugin> plugins=new HashSet<>();
```

に、 *URLClassLoader* をインスタンスし、クラスをして、プラグインインタフェースをするクラスをインスタンスします。

```
URLClassLoader urlClassLoader=new URLClassLoader(urls.toArray(new URL[urls.size()]));
classes.forEach(className->{
    try
    {
        Class
        cls=urlClassLoader.loadClass(className.replaceAll("/", ".").replace(".class", ""));
        //transforming to binary name
        Class[] interfaces=cls.getInterfaces();
        for(Class interface:interfaces)
        {
            if(interface.equals(Plugin.class)) //checking presence of Plugin interface
            {
                Plugin plugin=(Plugin) cls.newInstance(); //instantiating the Plugin
                plugins.add(plugin);
                break;
            }
        }
    }
    catch (Exception e){e.printStackTrace();}
});
```

に、プラグインのメソッドをびすことで、プラグインのメソッドをすることができます。

```
if(!plugins.isEmpty())loadedPlugins.getChildren().add(new Label("Loaded plugins:"));
plugins.forEach(plugin -> {
    plugin.initialize();
    loadedPlugins.getChildren().add(new Label(plugin.name()));
});
```

MainApplicationのコード

```
package main;
public class MainApplication extends Application
{
    static HashSet<Plugin> plugins=new HashSet<>();
    @Override
    public void start(Stage primaryStage) throws Exception
    {
        File pluginDirectory=new File("plugins");
        if(!pluginDirectory.exists())pluginDirectory.mkdir();
        File[] files=pluginDirectory.listFiles((dir, name) -> name.endsWith(".jar"));
        VBox loadedPlugins=new VBox(6);
        loadedPlugins.setAlignment(Pos.CENTER);
        if(files!=null && files.length>0)
        {
            ArrayList<String> classes=new ArrayList<>();
            ArrayList<URL> urls=new ArrayList<>(files.length);
            for(File file:files)
            {
                JarFile jar=new JarFile(file);
                jar.stream().forEach(jarEntry -> {
                    if(jarEntry.getName().endsWith(".class"))
                    {
                        classes.add(jarEntry.getName());
                    }
                });
                URL url=file.toURI().toURL();
                urls.add(url);
            }
            URLClassLoader urlClassLoader=new URLClassLoader(urls.toArray(new
            URL[urls.size()]));
            classes.forEach(className->{
                try
                {
                    Class
                    cls=urlClassLoader.loadClass(className.replaceAll("/", ".").replace(".class", ""));
                    Class[] interfaces=cls.getInterfaces();
                    for(Class intface:interfaces)
                    {
                        if(intface.equals(Plugin.class))
                        {
                            Plugin plugin=(Plugin) cls.newInstance();
                            plugins.add(plugin);
                            break;
                        }
                    }
                }
                catch (Exception e){e.printStackTrace();}
            });
            if(!plugins.isEmpty())loadedPlugins.getChildren().add(new Label("Loaded
```

```

plugins:"));
        plugins.forEach(plugin -> {
            plugin.initialize();
            loadedPlugins.getChildren().add(new Label(plugin.name()));
        });
    }
    Rectangle2D screenbounds=Screen.getPrimary().getVisualBounds();
    Scene scene=new
Scene(loadedPlugins,screenbounds.getWidth()/2,screenbounds.getHeight()/2);
    primaryStage.setScene(scene);
    primaryStage.show();
}
public static void main(String[] a)
{
    launch(a);
}
}

```

2つのプラグインをしましょう。らかに、プラグインのソースはのモジュールになければなりません。

```

package plugins;

import main.Plugin;

public class FirstPlugin implements Plugin
{
    //this plugin has default behaviour
}

```

2のプラグイン

```

package plugins;

import main.Plugin;

public class AnotherPlugin implements Plugin
{
    @Override
    public void initialize() //overridden to show user's home directory
    {
        System.out.println("User home directory: "+System.getProperty("user.home"));
    }
}

```

これらのプラグインは、のJarファイルにパッケージするがあります。このプロセスは、IDEやのツールによってなります。

Jarsが'プラグイン'にされると、*MainApplication*はそれらをしてなクラスをインスタンスします。

オンラインでJavaプラグインシステムのをむ <https://riptutorial.com/ja/java/topic/7160/javaプラグインシステムの>

40: Java プリント サービス

き

Java Print Service APIは、サービスをしてをするためのをします。

IETF RFC 2911のインターネットプロトコルIPP1.1でされたにつくなをむ。

Examples

なサービスの

なすべてのサービスをするために、 `PrintServiceLookup` クラスをできます。どのようにてみましょう

```
import javax.print.PrintService;
import javax.print.PrintServiceLookup;

public class DiscoveringAvailablePrintServices {

    public static void main(String[] args) {
        discoverPrintServices();
    }

    public static void discoverPrintServices() {
        PrintService[] allPrintServices = PrintServiceLookup.lookupPrintServices(null, null);

        for (PrintService printService : allPrintServices) {
            System.out.println("Print service name: " + printService.getName());
        }
    }
}
```

このプログラムは、Windowsでされると、のようにされます。

```
Print service name: Fax
Print service name: Microsoft Print to PDF
Print service name: Microsoft XPS Document Viewer
```

デフォルトのサービスの

デフォルトのプリントサービスをするために、 `PrintServiceLookup` クラスをできます。をてみましょう::

```
import javax.print.PrintService;
import javax.print.PrintServiceLookup;
```

```
public class DiscoveringDefaultPrintService {

    public static void main(String[] args) {
        discoverDefaultPrintService();
    }

    public static void discoverDefaultPrintService() {
        PrintService defaultPrintService = PrintServiceLookup.lookupDefaultPrintService();
        System.out.println("Default print service name: " + defaultPrintService.getName());
    }

}
```

サービスからのジョブの

ジョブは、のサービスにかをするです。には、

- されるデータ **されるのを**
- のセット

しいプリントサービスインスタンスをした、プリントジョブのをすることができます。

```
DocPrintJob printJob = printService.createPrintJob();
```

DocPrintJob インターフェイスは、 print メソッドをし print。

```
printJob.print(doc, pras);
```

doc は Doc されるデータです。

そして、 pras がある PrintRequestAttributeSet セットインタフェース PrintRequestAttribute。 のは のとおりです。

- コピー1,2、
- きまたは
- モノクロ、カラー
- ドラフト、ノーマル、ハイ
- 、など
- 々...

print メソッドは PrintException スローすることがあり PrintException。

されるの

Doc はインタフェースであり、Java Print Service API は SimpleDoc とばれるなをします。

すべての Doc インスタンスは、に2つのからりっています。

- データコンテンツメール、

- DocFlavor MIMEタイプ+クラスとばれるデータフォーマット。

Docオブジェクトをするに、ドキュメントをどこかからみむがあります。このでは、ディスクからのファイルをロードします。

```
FileInputStream pdfFileInputStream = new FileInputStream("something.pdf");
```

これで、たちのコンテンツとするDocFlavorをするがありDocFlavor。 DocFlavorクラスには、もなタイプのデータをすがDocFlavor。 INPUT_STREAM.PDFしましょう

```
DocFlavor pdfDocFlavor = DocFlavor.INPUT_STREAM.PDF;
```

さて、 SimpleDocしいインスタンスをすることができます

```
Doc doc = new SimpleDoc(pdfFileInputStream, pdfDocFlavor , null);
```

docオブジェクトをジョブにできるようになりました サービスからのジョブのを。

の

によっては、のいくつかのをするがあります。々はそれをとぶ。

のはのとおりです。

- コピー1,2、
- きまたは
- モノクロ、カラー
- ドラフト、ノーマル、ハイ
- 、など
- 々...

それらのうちの1つをするに、それぞれのにどのようながあるかをるには、まずのセットをするがあります。

```
PrintRequestAttributeSet pras = new HashPrintRequestAttributeSet();
```

これでできます。いくつかのがあります

```
pras.add(new Copies(5));
pras.add(MediaSize.ISO_A4);
pras.add(OrientationRequested.PORTRAIT);
pras.add(PrintQuality.NORMAL);
```

prasオブジェクトをジョブにできるようになりました サービスからのジョブのを。

リッスンジョブの

ほとんどのクライアントでは、ジョブがしたかしたかをるのににです。

Java Print Service APIは、これらのシナリオにするをるためのいくつかのをします。々がしななければならないのは、

- `PrintJobListener` インタフェースのをし、
- このをジョブにします。

ジョブのがわると、されます。たちはかかなことをうことができます。えば

- ユーザインタフェースをし、
- のビジネスプロセスをし、
- データベースにかをし、
- またはにログにします。

のでは、すべてのジョブのステータスをしします

```
import javax.print.event.PrintJobEvent;
import javax.print.event.PrintJobListener;

public class LoggerPrintJobListener implements PrintJobListener {

    // Your favorite Logger class goes here!
    private static final Logger LOG = Logger.getLogger(LoggerPrintJobListener.class);

    public void printDataTransferCompleted(PrintJobEvent pje) {
        LOG.info("Print data transfer completed ; ) ");
    }

    public void printJobCompleted(PrintJobEvent pje) {
        LOG.info("Print job completed =) ");
    }

    public void printJobFailed(PrintJobEvent pje) {
        LOG.info("Print job failed =( ");
    }

    public void printJobCanceled(PrintJobEvent pje) {
        LOG.info("Print job canceled :| ");
    }

    public void printJobNoMoreEvents(PrintJobEvent pje) {
        LOG.info("No more events to the job ");
    }

    public void printJobRequiresAttention(PrintJobEvent pje) {
        LOG.info("Print job requires attention :O ");
    }
}
```

に、のように、ジョブのに、ジョブのリスナーをジョブにすることができます。

```
DocPrintJob printJob = printService.createPrintJob();
```



```
printJob.addPrintJobListener(new LoggerPrintJobListener());

printJob.print(doc, pras);
```

PrintJobEvent pje

すべてのメソッドに `PrintJobEvent pje` があることにしてください。このではのためにしていませんが、ステータスをべるためにすることができます。えば

```
pje.getPrintJob().getAttributes();
```

`PrintJobAttributeSet` オブジェクトインスタンスをし、それぞれにして for each way をできます。

じをするの

じをするためのもう1つのオプションは、 `PrintJobAdapter` クラスをすることですがすように、 `PrintJobListener` アダプタです。インタフェースをするには、それらのすべてをにするがあります。こののは、たちがむメソッドだけをオーバーライドするがあることです。どのようにするかてみましょう

```
import javax.print.event.PrintJobEvent;
import javax.print.event.PrintJobAdapter;

public class LoggerPrintJobAdapter extends PrintJobAdapter {

    // Your favorite Logger class goes here!
    private static final Logger LOG = Logger.getLogger(LoggerPrintJobAdapter.class);

    public void printJobCompleted(PrintJobEvent pje) {
        LOG.info("Print job completed =) ");
    }

    public void printJobFailed(PrintJobEvent pje) {
        LOG.info("Print job failed =( ");
    }
}
```

のメソッドだけをオーバーライドすることにしてください。

インタフェースの `PrintJobListener` するとじで、プリントジョブにプリントをするにリスナーをします。

```
printJob.addPrintJobListener(new LoggerPrintJobAdapter());

printJob.print(doc, pras);
```

オンラインでJavaプリントサービスをむ <https://riptutorial.com/ja/java/topic/10178/javaプリントサービス>

41: Java メモリモデル

Javaメモリモデルは、あるスレッドが他のスレッドによってわかれたメモリきみのをることがされるをするJLSのセクションです。このエディションのセクションは、「JLS 17.4メモリモデル」 [Java 8](#)、[Java 7](#)、[Java 6](#)

Java 5では、Javaメモリモデルのなしがわれました。これはとりわけ `volatile` のがしました。それ、メモリモデルはにされていません。

Examples

メモリモデルのづけ

のをえてみましょう。

```
public class Example {
    public int a, b, c, d;

    public void doIt() {
        a = b + 1;
        c = d + 1;
    }
}
```

このクラスがシングルスレッドのアプリケーションでされている、なるいはどおりになります。
えは

```
public class SingleThreaded {
    public static void main(String[] args) {
        Example eg = new Example();
        System.out.println(eg.a + ", " + eg.c);
        eg.doIt();
        System.out.println(eg.a + ", " + eg.c);
    }
}
```

されます

```
0, 0
1, 1
```

"main"スレッドがるり、`main()`メソッドと`doIt()`メソッドの`doIt()`は、ソースコードでかれたでされます。これは、JavaJLSのなです。

に、マルチスレッドアプリケーションでされるじクラスについてえてみましょう。

```
public class MultiThreaded {
    public static void main(String[] args) {
```

```

final Example eg = new Example();
new Thread(new Runnable() {
    public void run() {
        while (true) {
            eg.doIt();
        }
    }
}).start();
while (true) {
    System.out.println(eg.a + ", " + eg.c);
}
}
}

```

これはをしますか

、JLSによれば、これがされることをすることはできません

- おそらく $0, 0$ までのがされます。
- に $N, N, N, N + 1$ ます。
- $N + 1, N$ ようながされることがあります。
- には、あなたもいることをされるがあります $0, 0$ が1にけます。

1 - には、`println`ステートメントのにより、あるのせぬとメモリキャッシュのフラッシュがするがあります。これは、のをきこすいくつかのをすがあります。

では、どうやってするのですか

りてのべえ

しないののあるの1つは、JITコンパイラが`doIt()`メソッドのをしたことです。JLSでは、ステートメントがのスレッドのからにされるようにするがあります。この、`doIt()`メソッドのコードでは、これら2つのステートメントのなべえのをすることはできません。これは、JITコンパイラがされることをします。

それはなぜでしょうか

ななハードウェアでは、は、パイプラインをしてされ、のがなるにあることをにする。のフェーズのにはのものよりがかかり、メモリにがかかるがあります。スマートコンパイラは、オーバーラップのをにするをけることによって、パイプラインのスループットをすることができます。これにより、ステートメントのがでされるがあります。JLSは、のスレッドのからののをえないように、これをします。

メモリキャッシュの

2にえられるはメモリキャッシングのです。なコンピュータアーキテクチャでは、プロセッサにはさなレジスタセットとのメモリがあります。レジスタへのアクセスはメインメモリへのアクセスよりもはるかにです。のアーキテクチャでは、レジスタよりもいがメインメモリよりもいメモリ

キャッシュがする。

コンパイラはのコピーをレジスタまたはメモリキャッシュにしようとするのでこれをします。は、メインメモリにフラッシュされるように、またはメモリからみされるはありませんがないは、これをやっていないのなパフォーマンスのがあります。JLSがメモリをのスレッドからえるがない、Java JITコンパイラはメモリのみみときみをする「みみバリア」と「きみバリア」をしないがあります。もう、これをする事によるパフォーマンスのはです。

な

これまでのところ、JLSはJITコンパイラがメモリをべえたりしたりすることでシングルスレッドコードをするコードをできるようになりました。しかし、のスレッドがメインメモリののをできるとどうなりますか

そのえは、のスレッドは、Javaステートメントのコードについてとわれるをするがあるということです。これにするは、なをすることです。な3つのアプローチはのとおりです。

- なmutexとsynchronizedされたの。
- volatileの。
- よりいいレベルのサポートをする。えは、`java.util.concurrent`パッケージのクラス

しかし、これでも、がなと、あなたがりにできるをすることがです。これは、Javaメモリモデルがするです。

メモリモデル

Javaメモリモデルは、あるスレッドがのスレッドによってわれたメモリきみのをることがされるをするJLSのセクションです。メモリモデルはななさでされており、としてするためにはでながです。しかし、なは、のが、あるスレッドによるのきみと、のスレッドによるじののみみとのに「こるbefore-before」をりすことである。「こる」がする、JITコンパイラは、みりがきみによってきまれたをにするコードをすることをけられています。

このことをまえれば、Javaプログラムのメモリコヒーレンシーをし、これがすべてのプラットフォームでであるかどうかをすることができます。

のにこること

は、Javaのされたバージョンです。よりいいのためには、をむがあります。

は、メモリモデルのであり、メモリのをしてすることができます。JLSがうように [JLS 17.4.5](#)

「2つのアクションは、なによってすることができます。あるアクションがこった、のアクションがこると、のアクションは2のアクションのにされ、けられます。

これはをするのでしょうか

のがするアクションは、 [JLS 17.4.2](#)でされています。でされている5のアクションがあります。

- みをみます。
- きみをきむ。
- アクション
 - みをみます。
 - きみをきむ。
 - ロック。モニターのロック
 - ロックをします。モニターのロックをする。
 - スレッドのとのアクション。
 - スレッドをするアクション、またはスレッドがしたことをするアクション。
- アクション。プログラムがされるにするをつアクション。
- スレッドのアクション。これらは、あるのループのをモデルします。

プログラムと

これらの2つの [JLS 17.4.3](#)および[JLS 17.4.4](#) は、Javaのステートメントのをします

プログラムは、のスレッドでののをする。

は、によってされた2つのステートメントのステートメントのをします。

- モニターのロックアクションは、そのモニターのすべてののロック・アクションとします。
- へのきみは、のすべてのスレッドによるじのみりとします。
- スレッドをするアクションつまり `Thread.start()` へのびし `Thread.start()` は、するスレッドのアクションつまりスレッドの `run()` メソッドのびしとします。
- フィールドのデフォルトのは、スレッドののアクションとします。これについては、JLSをしてください。
- スレッドののアクションは、をしたのスレッドのアクションとします。 `true` をす `join()` びしや `isTerminated()` びしのりなど `true` 。
- あるスレッドがのスレッドにりむと、そのスレッドがしたことのをスレッドがしたで、のスレッドのりみびしがします。

にこる

このけ [JLS 17.4.5](#) は、メモリミガのメモリリでえることがされるかどうかをするものです。

には、のみり_vへのきみをすることがされて_vびだけ`write(v)` `read(v)` とにきみはありません_v。きみがする、`read(v)` はのきみではなく、そのをることができます。

をするルールはのとおりです。

- ルール**1**のにこること - `x`と`y`がじスレッドのアクションであり、`x`がプログラムで`y`のにる、`x`は`y`のにします。
- ルール**2**よりのこと - オブジェクトのコンストラクタのわりからそのオブジェクトのファイナライザのまでのエッジがあります。
- ルール**3**のにこる - アクション`x`がのアクション`y` とする、`x`は `y`のにする。
- ルール**4**のに こること - `x` がきると、 `y`と`y` がこる、つまり`z`のに`x` がこり、 `z`のにこります。

また、Javaのライブラリの々なクラスをとてがされています。これは、がどのようにされるかをにるなく、 らかのでこることをするとできます。

こること - いくつかのにかされるに

たちは、きみが、のみみにされていることをするためにをするをすために、いくつかのをします。

シングルスレッドコード

どおり、きみは、シングルスレッドプログラムでののみりではにされます。

```
public class SingleThreadExample {
    public int a, b;

    public int add() {
        a = 1;           // write(a)
        b = 2;           // write(b)
        return a + b;    // read(a) followed by read(b)
    }
}
```

のにルール**1**のに

1. `write(a)` アクションがこるに、 `write(b)` アクション。
2. `write(b)` アクションは、 `read(a)` アクションのにします。
3. `read(a)` アクションは、 `read(a)` アクションのにします。

ルール4のにこったこと

4. `write(a)` がこるに、 `write(b)` および `write(b)` がこるに、 `read(a)` **IMPLIES** `write(a)` し、に `read(a)`
5. `write(b)` のに、こる `read(a)` **AND** `read(a)` がこるに、 `read(b)` **IMPLIES** `write(b)` `read(b)`
6. `write(a)` `read(a)` があることをし `a + b` はしいをすることがされて。 `a`
7. `write(b)` `read(b)` があることをし `a + b` はしいをすることがされて `b`。

2つのスレッドをつにおける 'volatile'のるい

のサンプルコードをして、メモリモデルの `volatile`のる。

```
public class VolatileExample {
    private volatile int a;
    private int b;          // NOT volatile

    public void update(int first, int second) {
        b = first;          // write(b)
        a = second;          // write-volatile(a)
    }

    public int observe() {
        return a + b;        // read-volatile(a) followed by read(b)
    }
}
```

まず、2つのスレッドをむのステートメントのシーケンスをえてみましょう。

1. `VolatileExample`のインスタンスがされます。それをびす `ve`、
2. `ve.update(1, 2)` が1つのスレッドでびされ、
3. `ve.observe()` はのスレッドでびされます。

のにルール1のに

1. `write(a)` アクションは、 `volatile-write(a)` アクションのにします。
2. `volatile-read(a)` アクションは、 `read(b)` アクションのにします。

ルール2よりもにこったこと

3. 1スレッドの `volatile-write(a)` アクションは、2スレッドの `volatile-read(a)` アクションのにします。

ルール4のにこったこと

4. 1スレッドの `write(b)` アクションは、2スレッドの `read(b)` アクションのにします。

いえれば、こののシーケンスでは、2のスレッドが1のスレッドによってられた `b` へのるることがされる。しかし、それはまた、りてにらかであるである `update` は、のでした、または `observe()` メソッドは、みる `b` にこる、にチェーンがされることになります。 `a` 2のスレッドの `volatile-read(a)` が1

のスレッドの `volatile-write(a)` にいていない、チェーンもします。

チェーンがれている、`observe()` にしいの `b` がされるというはありません。

3つのスレッドで

のに3のスレッドをします。

1. `VolatileExample` のインスタンスがされます。それをびす `ve` 、
2. 2つのスレッドが `update` びします。
 - `ve.update(1, 2)` が1つのスレッドでびされ、
 - `ve.update(3, 4)` が2のスレッドでびされ、
3. `ve.observe()` はその3のスレッドでびされます。

これをにするには、スレッド1とスレッド2のステートメントのなインターリーブをすべてするがあります。わりに、それらのうちの2つだけをします。

シナリオ1 - `update(1, 2)` が `update(3, 4)` してこのシーケンスをします

```
write(b, 1), write-volatile(a, 2)    // first thread
write(b, 3), write-volatile(a, 4)    // second thread
read-volatile(a), read(b)           // third thread
```

この、`write(b, 3)` から `read(b)` までしてこっていないがあることはにかります。さらに、`b` へのきみはありません。したがって、このシナリオでは、3のスレッドは `b` が `3` をつことをします。

シナリオ2 - `update(1, 2)` と `update(3, 4)` がなっており、のようにインターリーブされているとします

。

```
write(b, 3)                          // second thread
write(b, 1)                          // first thread
write-volatile(a, 2)                 // first thread
write-volatile(a, 4)                 // second thread
read-volatile(a), read(b)            // third thread
```

さて、`write(b, 3)` から `read(b)` までののがあるに、のスレッドによってされるする `write(b, 1)` アクションがあります。これは、`read(b)` がどのをするかをできないことをします。

これは、にられたをいて、の `volatile` をするために `volatile` にすることができないことをしています。

メモリモデルをするをする

メモリモデルはしにくく、するのがしいです。マルチスレッドコードのをするがあるにですが、するすべてのマルチスレッドアプリケーションにしてこのをうはありません。

Javaでコードをするのにのプリンシパルをすれば、のにるはほとんどなくなります。

- であれば、のデータをしてください。しくされたのクラスはスレッドセーフであり、のクラスとにするとときにスレッドセーフのがしません。
- 「でない」をし、する。
- プリミティブミューテックスまたは`Lock`スレッドセーフ¹であるはオブジェクトへのアクセスをするためにオブジェクトを。
- マネージスレッドをするのではなく、`Executor / ExecutorService`またはfork join frameworkをしてください。
- `wait / notify / notifyAll`をするわりに、なロック、セマフォ、ラッチ、バリアをする
`java.util.concurrent`クラスをしてください。
- コレクションではなく、マップ、セット、リスト、キューおよびキューの
`java.util.concurrent`バージョンをします。

なは、「の」をするのではなく、Javaのみみライブラリをすることです。あなたがそれらをにするならば、らがいていることにすることができます。

1 - すべてのオブジェクトがスレッドセーフであるはありません。たとえば、1つまたはのオブジェクトがスレッドされている つまり、1つのスレッドのみがアクセスな、スレッドセーフティはありません。

オンラインでJavaメモリモデルをむ <https://riptutorial.com/ja/java/topic/6829/javaメモリモデル>

42: Java メモリ

Javaでは、オブジェクトはヒープにリテラブル、ヒープメモリはガベージコレクションによってされます。アプリケーションプログラムは、Javaオブジェクトをにすることはできません。

Javaガベージコレクションのは、[ガベージコレクション](#)のしています。のでは、ファイナライズ、でガベージコレクタをトリガする、およびストレージリークのについてしています。

Examples

Javaオブジェクトは`finalize`メソッドをすることができます。このメソッドは、Javaがオブジェクトのメモリをするにびされます。、のようになります。

```
public class MyClass {  
  
    //Methods for the class  
  
    @Override  
    protected void finalize() throws Throwable {  
        // Cleanup code  
    }  
}
```

しかし、Javaファイナライズのにはいくつかのがあります。

- Javaは、`finalize()`メソッドがいつばれるかをしません。
- Javaは、のアプリケーションのに`finalize()`メソッドがしばらくびされることさえしません。
- されるのは、オブジェクトがされるにメソッドがびされるということだけです。オブジェクトがされたです。

のは、`finalize`メソッドをしてタイムリーにするがあるクリーンアップまたはそののアクションをすることはいえであることをします。ファイナライズににすると、ストレージリークやメモリリークなどのがするがあります。

するに、ファイナライズがにはいであるはほとんどありません。

ファイナライザは1だけされます

、オブジェクトはファイナライズされたにされます。しかし、これはにこるわけではありません。の1をえてみましょう。

```
public class CaptainJack {  
    public static CaptainJack notDeadYet = null;  
  
    protected void finalize() {
```

```

        // Resurrection!
        notDeadYet = this;
    }
}

```

CaptainJackのインスタンスがになり、ガベージコレクタがそれをしようとすると、`finalize()`メソッドはインスタンスへのを`notDeadYet`に`notDeadYet`ます。これにより、もうインスタンスにできるようになり、ガベージコレクタはそれをしません。

キャプテンジャックはですか

いいえ。

キャッチは、JVMはオブジェクトのファイナライザをだけするということです。 `notDeadYet`に`null`を`notDeadYet`と、`resurrect`されたインスタンスがびになるため、ガベージコレクタはオブジェクトにして`finalize()`をびしません。

1 - https://en.wikipedia.org/wiki/Jack_Harknessをしてください。

GCのトリガ

ガベージコレクタをびすことによってでトリガすることができます

```
System.gc();
```

ただし、コールがったときにJavaがガベージコレクタをしたことをするものではありません。このメソッドは、JVMJavaマシンにガベージコレクタをさせたいとに「」しますが、するようしません。

にガベージコレクションをでしようとするのはいとえられます。JVMは、`-XX:+DisableExplicitGC`オプションをしてすると、`System.gc()`へのびしをにできます。`System.gc()`びしてガベージコレクションをトリガすると、JVMがするのガベージコレクタののガベージ/オブジェクトがされるがあります。

ガベージコレクション

C++のアプローチ - と

C++のようなでは、アプリケーションプログラムは、にりてられたメモリによってされるメモリをします。`new`をしてC++ヒープにオブジェクトがされると、そのオブジェクトをするための`delete`をしてするがあります。

- プログラムがオブジェクトを`delete`することをれて、にそれをれると、するメモリはアプリケーションにわれま。こののはメモリリークであり、メモリリークがすぎると、アプリケーションがメモリをいやすくなり、にクラッシュするがあります。

•

、アプリケーションがオブジェクトを`delete`しようとした、またはオブジェクトがされたにそのオブジェクトをすると、メモリのによりアプリケーションがクラッシュするがあります

なC++プログラムでは、`new`と`delete`をしてメモリをするとが掛かることがあります。、メモリはバグのソースです。

Javaのアプローチ - ガベージコレクション

Javaにはなるアプローチがあります。な`delete`ではなく、Javaはガベージコレクションと呼ばれるメカニズムをして、になったオブジェクトがするメモリをします。Javaシステムは、されるオブジェクトをつけるがあります。このタスクは、ガベージコレクタ GCと呼ばれるコンポーネントによってされます。

Javaプログラムのにいつでも、のすべてのオブジェクトのセットを2つのなるサブセット¹にできます。

- なオブジェクトは、JLSによってのようになされます。

なオブジェクトは、のライブスレッドからのなにおいてアクセスなのオブジェクトである。

には、これはスコープのローカルからまるのや、あるコードがオブジェクトにできる`static`があることをします。

- オブジェクトは、のようになできないオブジェクトです。

なオブジェクトはガベージコレクションのとなります。これがガベージコレクションされるわけではありません。には

- なオブジェクトはになとちにされません¹。
- なオブジェクトはガベージコレクションされることはありません。

Javaは、オブジェクトをいつするかをするために、JVMにくのさをえます。また、にはなオブジェクトをどのようにするかについて、JVMがになことをします。

JLSがすることの1つは、なオブジェクトがガベージコレクションされないことです。

オブジェクトがになるとどうなるか

まずに、オブジェクトがになると、にもこりません。ガベージコレクタがされ、それは、オブジェクトがであることをしたときにのみこります。さらに、GCがなオブジェクトをすべてしないことがです。

GCがオブジェクトをすると、のイベントがするがあります。

1. オブジェクトをする`Reference`オブジェクトがある、それらはオブジェクトがされるにクリアされます。
2. オブジェクトがファイナライズなは、ファイナライズされます。これは、オブジェクトがされるにします。
3. オブジェクトはすることができ、それがめるメモリはすることができます。

のイベントがするのあるなシーケンスがありますが、ガベージコレクタはのののオブジェクトのをするはありません。

なオブジェクトとなオブジェクトの

のサンプルクラスをえてみましょう。

```
// A node in simple "open" linked-list.
public class Node {
    private static int counter = 0;

    public int nodeNumber = ++counter;
    public Node next;
}

public class ListTest {
    public static void main(String[] args) {
        test(); // M1
        System.out.println("Done"); // M2
    }

    private static void test() {
        Node n1 = new Node(); // T1
        Node n2 = new Node(); // T2
        Node n3 = new Node(); // T3
        n1.next = n2; // T4
        n2 = null; // T5
        n3 = null; // T6
    }
}
```

`test()` がびされたときにがきるかべてみましょう。ステートメントT1、T2、T3は`Node`オブジェクトをし、オブジェクトはそれぞれ`n1`、`n2`、`n3`をしてすべてです。ステートメントT4は、2の`Node`オブジェクトへののをの`Node`オブジェクトの`next`フィールドにりてます。これがすると、2つの`Node`は2つのパスでできます。

```
n2 -> Node2
n1 -> Node1, Node1.next -> Node2
```

ステートメントT5では、`n2`に`null`をし`null`。これにより、`Node2`のチェーンののものがし`Node2`が、2のものはられていないので、`Node2`はまだです。

ステートメントT6では、`n3`に`null`をし`null`。これにより、`Node3`チェーンが`Node3`、`Node3`できな

くなります。ただし、`Node1`と`Node2`に`n1`をしてです。

に、`test()`メソッドがってくると、ローカル`n1`、`n2`、`n3`はになり、したがつてもアクセスすることはできません。これにより、`Node1`と`Node2`のりのチェーンがられ、すべての`Node`オブジェクトにアクセスできなくなり、ガベージコレクションがになります。

1- これは、ファイナライズをするであり、`Reference`クラスです。に、Javaのはこれをうことができますが、これをうパフォーマンスコストはではありません。

ヒープ、**PermGen**、およびスタックサイズの

Javaマシンがするときには、ヒープをするきさとスレッドスタックのデフォルトサイズをるがあります。これらは、`java`コマンドのコマンドラインオプションをしてできます。Java 8よりのバージョンのJavaでは、ヒープのPermGenのサイズをすることもできます。

PermGenはJava 8でされており、PermGenサイズをしようとすると、オプションはされますメッセージ。

ヒープとスタックのサイズをにしない、JVMはバージョンとプラットフォームのでされるデフォルトをします。これにより、アプリケーションのメモリがなすぎたりすぎたりすることがあります。、これはスレッドスタックではありませんが、のメモリをするプログラムではになるがあります。

ヒープ、**PermGen**、およびデフォルトのスタックサイズの

のJVMオプションは、ヒープサイズをします。

- `-Xms<size>` - ヒープサイズをします。
- `-Xmx<size>` - ヒープサイズをします。
- `-XX:PermSize<size>` - `-XX:PermSize<size>` します。
- `-XX:MaxPermSize<size>` - PermGenのサイズをします。
- `-Xss<size>` - デフォルトのスレッドスタックサイズをします。

`<size>`パラメータはバイトでも、`k`、`m`または`g`をつこともできます。は、サイズをそれぞれキロバイト、メガバイト、およびギガバイトでします。

```
$ java -Xms512m -Xmx1024m JavaApp
$ java -XX:PermSize=64m -XX:MaxPermSize=128m JavaApp
$ java -Xss512k JavaApp
```

デフォルトサイズをつける

`-XX:+printFlagsFinal`オプションをすると、JVMをするにすべてのフラグのをできます。これは、ヒープサイズとスタックサイズののデフォルトをのようにするためにできます。

- Linux、Unix、Solaris、Mac OSXの

```
$ java -XX+ PrintFlagsFinal -version | grep -iE 'HeapSize | PermSize | ThreadStackSize'
```

- Windowsの

```
java -XX+ PrintFlagsFinal -version | findstr /i "HeapSize PermSize  
ThreadStackSize"
```

のコマンドのは、のようになります。

```
uintx InitialHeapSize          := 20655360      {product}  
uintx MaxHeapSize              := 331350016      {product}  
uintx PermSize                 = 21757952        {pd product}  
uintx MaxPermSize              = 85983232        {pd product}  
intx ThreadStackSize           = 1024            {pd product}
```

サイズはバイトでえられます。

Javaでのメモリリーク

ガベージコレクションのでは、Javaがメモリリークのをすることをしました。これはにはてはまりません。Javaプログラムはメモリリークをこすがありますが、リークのはかなりなります。

なオブジェクトがれる

のなスタックのをえてみましょう。

```
public class NaiveStack {  
    private Object[] stack = new Object[100];  
    private int top = 0;  
  
    public void push(Object obj) {  
        if (top >= stack.length) {  
            throw new StackException("stack overflow");  
        }  
        stack[top++] = obj;  
    }  
  
    public Object pop() {  
        if (top <= 0) {  
            throw new StackException("stack underflow");  
        }  
        return stack[--top];  
    }  
  
    public boolean isEmpty() {  
        return top == 0;  
    }  
}
```

オブジェクトを`push`てすぐに`pop`すると、`stack`のオブジェクトへのがききし`stack`。

スタックのロジックは、そのをAPIのクライアントにすことができないことをします。オブジェ

クトがポップされている、ライブスレッドからのななではアクセスできないことをできます。は、のJVMではこれをできないということです。のJVMは、がかどうかをするにプログラムのロジックをしません。めはではありません

しかし、がどういふなのかをにすれば、`NaiveStack`がすべきオブジェクトに「ぶらがっている」があることは確かです。それはメモリリークです。

この、はです。

```
public Object pop() {
    if (top <= 0) {
        throw new StackException("stack underflow");
    }
    Object popped = stack[--top];
    stack[top] = null;          // Overwrite popped reference with null.
    return popped;
}
```

キャッシュはメモリリークがあります

サービスのパフォーマンスをさせるためのなは、をキャッシュすることです。なとそののを、キャッシュと呼ばれるメモリのデータにするというえです。に、がわれるたびに、キャッシュのをルックアップします。ルックアップがすると、するされたがされます。

このは、にすればにです。しかし、しくされていないと、キャッシュがメモリリークになるがあります。のをえてみましょう。

```
public class RequestHandler {
    private Map<Task, Result> cache = new HashMap<>();

    public Result doRequest(Task task) {
        Result result = cache.get(task);
        if (result == null) {
            result = doRequestProcessing(task);
            cache.put(task, result);
        }
        return result;
    }
}
```

このコードのは、`doRequest`びし`doRequest`キャッシュにしいエントリがされるがあるに、それらをするものがないことです。サービスがになるタスクをしている、キャッシュはにすべてのなメモリをします。これはメモリリークのです。

これをする1つのアプローチは、サイズのキャッシュをし、キャッシュがをえたときにいエントリをすることです。もされたエントリを`WeakHashMap`はいです。もうつのアプローチは`WeakHashMap`をってキャッシュをして、ヒープがいっぱいになってもJVMがキャッシュエントリを`WeakHashMap`できるようにすることです。

オンラインでJavaメモリをむ <https://riptutorial.com/ja/java/topic/2804/javaメモリ>

43: JavaマシンJVM

Examples

これらはです。

JVMは、コンピューティングマシンまたはマシンであり、RAMにします。Javaバイトコードをネイティブマシンコードにする、プラットフォームにしないをえています。JavacはJavaコードをバイトコードにコンパイルするJavaコンパイラです

JavaプログラムはJVMでされ、JVMはとなるマシンにマップされます。JDKのプログラミングツールの1つです。

*Byte code*はすべてのプラットフォームでされるプラットフォームにしないコードであり、*Machine code*はWindowsまたはLinuxなどのプラットフォームでのみされるプラットフォームのコードであり、にします。

のコンポーネント -

- Class Loder - .classファイルをRAMにロードします。
- バイトコードベリファイア - コードにアクセスがないかどうかをします。
- エンジン - バイトコードをなマシンコードにします。
- JITJIT - JITはJVMのです。JVMのパフォーマンスをさせました。にJavaバイトコードをにコンパイルまたはネイティブマシンコードにします。

オンラインでJavaマシンJVMをむ <https://riptutorial.com/ja/java/topic/8110/javaマシン-jvm->

44: Java

き

は、をつですはでされます。Javaでは、`float` 4バイトと`double` 8バイトの2つのプリミティブがあります。このドキュメンテーションページは、Javaのでできるをしくしています。

Examples

の

`==`、`!=`、`<`などをして `float` または `double` をするときにはがです。これらは、のバイナリによってをします。えは

```
public class CompareTest {
    public static void main(String[] args) {
        double oneThird = 1.0 / 3.0;
        double one = oneThird * 3;
        System.out.println(one == 1.0);    // prints "false"
    }
}
```

`oneThird` さなめをしており、々はけたときに`oneThird`することによって3々はわずかになるをる`1.0`。

このなのは、で`double`と`float`をぜようとすると、よりくなります。えは

```
public class CompareTest2 {
    public static void main(String[] args) {
        float floatVal = 0.1f;
        double doubleVal = 0.1;
        double doubleValCopy = floatVal;

        System.out.println(floatVal);    // 0.1
        System.out.println(doubleVal);    // 0.1
        System.out.println(doubleValCopy); // 0.10000000149011612

        System.out.println(floatVal == doubleVal); // false
        System.out.println(doubleVal == doubleValCopy); // false
    }
}
```

`float`と`double`のJavaでされるは、のがあります。 `float`の、は23バイナリまたは8です。 `double`タイプの、52ビットまたは15の10です。さらに、によってはめがすることがあります。したがって、プログラムがをする、のためのデルタをするのがなです。2つののがデルタよりさい、それらはしいとみなされる。えは

```
if (Math.abs(v1 - v2) < delta)
```

```

public class DeltaCompareExample {

    private static boolean deltaCompare(double v1, double v2, double delta) {
        // return true iff the difference between v1 and v2 is less than delta
        return Math.abs(v1 - v2) < delta;
    }

    public static void main(String[] args) {
        double[] doubles = {1.0, 1.0001, 1.00000001, 1.0000000001, 1.00000000000001};
        double[] deltas = {0.01, 0.00001, 0.00000001, 0.0000000001, 0};

        // loop through all of deltas initialized above
        for (int j = 0; j < deltas.length; j++) {
            double delta = deltas[j];
            System.out.println("delta: " + delta);

            // loop through all of the doubles initialized above
            for (int i = 0; i < doubles.length - 1; i++) {
                double d1 = doubles[i];
                double d2 = doubles[i + 1];
                boolean result = deltaCompare(d1, d2, delta);

                System.out.println("'" + d1 + "' == '" + d2 + "' ? " + result);
            }

            System.out.println();
        }
    }
}

```

```

delta: 0.01
1.0 == 1.0001 ? true
1.0001 == 1.00000001 ? true
1.00000001 == 1.0000000001 ? true
1.0000000001 == 1.00000000000001 ? true

delta: 1.0E-5
1.0 == 1.0001 ? false
1.0001 == 1.00000001 ? false
1.00000001 == 1.0000000001 ? true
1.0000000001 == 1.00000000000001 ? true

delta: 1.0E-7
1.0 == 1.0001 ? false
1.0001 == 1.00000001 ? false
1.00000001 == 1.0000000001 ? true
1.0000000001 == 1.00000000000001 ? true

delta: 1.0E-10
1.0 == 1.0001 ? false
1.0001 == 1.00000001 ? false
1.00000001 == 1.0000000001 ? false
1.0000000001 == 1.00000000000001 ? false

delta: 0.0
1.0 == 1.0001 ? false
1.0001 == 1.00000001 ? false

```

```
1.00000001 == 1.0000000001 ? false
1.0000000001 == 1.0000000000000001 ? false
```

また、`double`と`float`プリミティブの`compare`のために、`boxing`の`static compare`メソッドをできます。例えば

```
double a = 1.0;
double b = 1.0001;

System.out.println(Double.compare(a, b)); //-1
System.out.println(Double.compare(b, a)); //1
```

に、どのデルタがにもかをするのはいいことがあります。にされているは、デルタをすることです。しかし、のとのさ、およびされたをっている、の、したがってデルタのになをいくことができます。こののをうにえられていたとしてられているなのがあります。

オーバーフローとアンダーフロー

データ

データは、の32ビットIEEE 754です。

Float オーバーフロー

なは`3.4028235e+38`、このをえると`Infinity`

```
float f = 3.4e38f;
float result = f*2;
System.out.println(result); //Infinity
```

Float アンダーフロー

は`1.4e-45f`で、このをと`0.0`されます。

```
float f = 1e-45f;
float result = f/1000;
System.out.println(result);
```

ダブルデータ

`double`データは、の64-bit IEEE 754です。

Double オーバーフロー

なは`1.7976931348623157e+308`。このをえると、`Infinity`

```
double d = 1e308;
double result=d*2;
System.out.println(result); //Infinity
```

Double アンダーフロー

は4.9e-324で、このをとると0.0されます。

```
double d = 4.8e-323;
double result = d/1000;
System.out.println(result); //0.0
```

の

は、'f'フラグをつString.formatをして10でフォーマットできます

```
//Two digits in fractional part are rounded
String format1 = String.format("%.2f", 1.2399);
System.out.println(format1); // "1.24"

// three digits in fractional part are rounded
String format2 = String.format("%.3f", 1.2399);
System.out.println(format2); // "1.240"

//rounded to two digits, filled with zero
String format3 = String.format("%.2f", 1.2);
System.out.println(format3); // returns "1.20"

//rounder to two digits
String format4 = String.format("%.2f", 3.19999);
System.out.println(format4); // "3.20"
```

は、DecimalFormatをして10としてフォーマットできます

```
// rounded with one digit fractional part
String format = new DecimalFormat("0.#").format(4.3200);
System.out.println(format); // 4.3

// rounded with two digit fractional part
String format = new DecimalFormat("0.##").format(1.2323000);
System.out.println(format); //1.23

// formatting floating numbers to decimal number
double dv = 123456789;
System.out.println(dv); // 1.23456789E8
String format = new DecimalFormat("0").format(dv);
System.out.println(format); //123456789
```

IEEEへのな

デフォルトでは、floatとdoubleは、IEEE 754ののににわない。では、これらののにするのをする
ことができます。にそれらがにであることをにする。

strictfpはこのをにします。それはこのようなそののクラス、インタフェース、メソッド、コン
ストラクタ、イニシャライザなど、クラス、インタフェース、またはメソッドにされ、そのにま
れるすべてのものにされるstrictfp、のはでなければなりません floatセットまたはdoubleセット

。これにより、そのようなのは、IEEE 754がするとにします。

すべてののはstrictfpスコープになくても、にです。

したがって、strictfpにはのコーナーケースののをさせるのをちます。また、をくすることもできます CPUがネイティブのをしてにをえないようにします。ただし、はすべてのプラットフォームでまったくじになります。それは、がスピードよりもであるプログラムのようなものにです。

```
public class StrictFP { // No strictfp -> default lenient
    public strictfp float strict(float input) {
        return input * input / 3.4f; // Strictly adheres to the spec.
        // May be less accurate and may be slower.
    }

    public float lenient(float input) {
        return input * input / 3.4f; // Can sometimes be more accurate and faster,
        // but results may not be reproducible.
    }

    public static final strictfp class Ops { // strictfp affects all enclosed entities
        private StrictOps() {}

        public static div(double dividend, double divisor) { // implicitly strictfp
            return dividend / divisor;
        }
    }
}
```

オンラインでJavaをむ <https://riptutorial.com/ja/java/topic/6167/java>

45: JAXB

き

JAXBまたはJAXB [Java Architecture for XML Binding](#) は、JavaがJavaクラスをXMLにマップできるソフトウェアフレームワークです。このページでは、JAXBのを、にJavaオブジェクトをXMLフォーマットにマーシャリングおよびマーシャリングするためにされるそのにするなをしてします。

- JAXB.marshallObject、fileObjOfXML;
- オブジェクトobj = JAXB.unmarshallfileObjOfXML、className;

パラメーター

パラメータ	
fileObjOfXML	XMLファイルのFileオブジェクト
クラス	.classをつクラスの

JDKでなXJCツールをすると、XMLスキーマ .xsdファイルでされたXMLのJavaコードがにされます [XJCトピック](#)を。

Examples

XML ファイルをするオブジェクトをマーシャリングする

```
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class User {

    private long userID;
    private String name;

    // getters and setters
}
```

アノテーション `XmlRootElement` をすることにより、クラスをXMLファイルのルートとしてマークできます。

```
import java.io.File;
import javax.xml.bind.JAXB;
```



```

public class XMLCreator {
    public static void main(String[] args) {
        User user = new User();
        user.setName("Jon Skeet");
        user.setUserID(8884321);

        try {
            JAXB.marshal(user, new File("UserDetails.xml"));
        } catch (Exception e) {
            System.err.println("Exception occurred while writing in XML!");
        } finally {
            System.out.println("XML created");
        }
    }
}

```

`marshal()` は、オブジェクトのコンテンツをXMLファイルにきむためにされます。ここでは、`user` オブジェクトとしての `File` オブジェクトがとして `marshal()` されます。

がすると、`class-path`にの `UserDetails.xml` というのXMLファイルがされます。

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<user>
  <name>Jon Skeet</name>
  <userID>8884321</userID>
</user>

```

XML ファイルのみみアンマーシャリング

の `UserDetails.xml` というのXMLファイルのみむには

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<user>
  <name>Jon Skeet</name>
  <userID>8884321</userID>
</user>

```

のように、`User.java` というのPOJOクラスが `User.java`

```

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class User {

    private long userID;
    private String name;

    // getters and setters
}

```

ここでは、XMLノードによってクラスをしました。それらをマップするために、たちはクラスの `XmlRootElement` をします。

```

public class XMLReader {
    public static void main(String[] args) {
        try {
            User user = JAXB.unmarshal(new File("UserDetails.xml"), User.class);
            System.out.println(user.getName());    // prints Jon Skeet
            System.out.println(user.getUserID()); // prints 8884321
        } catch (Exception e) {
            System.err.println("Exception occurred while reading the XML!");
        }
    }
}

```

`unmarshal()` メソッドは、XMLファイルのにされます。2つのとしてXMLファイルとクラスをります。に、オブジェクトのゲッターメソッドをしてデータをすることができます。

XmlAdapterをしてのxmlをする

なXMLフォーマットがJavaオブジェクトモデルとなる、XmlAdapterをしてモデルオブジェクトをxmlフォーマットオブジェクトにすることができます。このは、フィールドのをつのにフィールドのをするをしています。

```

public class XmlAdapterExample {

    @XmlAccessorType(XmlAccessType.FIELD)
    public static class NodeValueElement {

        @XmlAttribute(name="attrValue")
        String value;

        public NodeValueElement() {
        }

        public NodeValueElement(String value) {
            super();
            this.value = value;
        }

        public String getValue() {
            return value;
        }

        public void setValue(String value) {
            this.value = value;
        }
    }

    public static class ValueAsAttrXmlAdapter extends XmlAdapter<NodeValueElement, String> {

        @Override
        public NodeValueElement marshal(String v) throws Exception {
            return new NodeValueElement(v);
        }

        @Override
        public String unmarshal(NodeValueElement v) throws Exception {
            if (v==null) return "";
            return v.getValue();
        }
    }
}

```

```

    }
}

@XmlRootElement(name="DataObject")
@XmlAccessorType(XmlAccessType.FIELD)
public static class DataObject {

    String elementWithValue;

    @XmlJavaTypeAdapter(value=ValueAsAttrXmlAdapter.class)
    String elementWithAttribute;
}

public static void main(String[] args) {
    DataObject data = new DataObject();
    data.elementWithValue="value1";
    data.elementWithAttribute ="value2";

    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    JAXB.marshal(data, baos);

    String xmlString = new String(baos.toByteArray(), StandardCharsets.UTF_8);

    System.out.println(xmlString);
}
}

```

フィールド/プロパティXMLマッピング@XmlAccessorType

`@XmlAccessorType` は、フィールド/プロパティをXMLにシリアルするかどうかをします。

`@XmlElement`、`@XmlAttribute` `@XmlElement`、または`@XmlTransient` フィールドとメソッドは、デフォルトよりもされることにしてください。

```

public class XmlAccessTypeExample {

    @XmlAccessorType(XmlAccessType.FIELD)
    static class AccessorExampleField {
        public String field="value1";

        public String getGetter() {
            return "getter";
        }

        public void setGetter(String value) {}
    }

    @XmlAccessorType(XmlAccessType.NONE)
    static class AccessorExampleNone {
        public String field="value1";

        public String getGetter() {
            return "getter";
        }

        public void setGetter(String value) {}
    }

    @XmlAccessorType(XmlAccessType.PROPERTY)

```

```

static class AccessorExampleProperty {
    public String field="value1";

    public String getGetter() {
        return "getter";
    }

    public void setGetter(String value) {}
}

@XmlAccessorType(XmlAccessType.PUBLIC_MEMBER)
static class AccessorExamplePublic {
    public String field="value1";

    public String getGetter() {
        return "getter";
    }

    public void setGetter(String value) {}
}

public static void main(String[] args) {
    try {
        System.out.println("\nField:");
        JAXB.marshal(new AccessorExampleField(), System.out);
        System.out.println("\nNone:");
        JAXB.marshal(new AccessorExampleNone(), System.out);
        System.out.println("\nProperty:");
        JAXB.marshal(new AccessorExampleProperty(), System.out);
        System.out.println("\nPublic:");
        JAXB.marshal(new AccessorExamplePublic(), System.out);
    } catch (Exception e) {
        System.err.println("Exception occurred while writing in XML!");
    }
}

} // outer class end

```

```

Field:
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<accessorExampleField>
    <field>value1</field>
</accessorExampleField>

None:
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<accessorExampleNone/>

Property:
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<accessorExampleProperty>
    <getter>getter</getter>
</accessorExampleProperty>

Public:
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<accessorExamplePublic>
    <field>value1</field>
    <getter>getter</getter>
</accessorExamplePublic>

```

フィールド/プロパティXMLマッピング

`@XmlElement`、`@XmlAttribute`または`@XmlTransient`、パッケージの`javax.xml.bind.annotation`プログラマがシリアライズされるべきであるとどのようにマークされたフィールドまたはプロパティをすることができます。

```
@XmlAccessorType(XmlAccessType.NONE) // we want no automatic field/property marshalling
public class ManualXmlElementsExample {

    @XmlElement
    private String field="field value";

    @XmlAttribute
    private String attribute="attr value";

    @XmlAttribute(name="differentAttribute")
    private String oneAttribute="other attr value";

    @XmlElement(name="different name")
    private String oneName="different name value";

    @XmlTransient
    private String transientField = "will not get serialized ever";

    @XmlElement
    public String getModifiedTransientValue() {
        return transientField.replace(" ever", "", unless in a getter");
    }

    public void setModifiedTransientValue(String val) {} // empty on purpose

    public static void main(String[] args) {
        try {
            JAXB.marshal(new ManualXmlElementsExample(), System.out);
        } catch (Exception e) {
            System.err.println("Exception occurred while writing in XML!");
        }
    }
}
```

のデータをするためのXmlAdapter インスタンスの

によっては、データののインスタンスをするがあります。レクリエーションはましくなく、`static`データのにはコードのいがあります。

`Unmarshaller`がするがある`XmlAdapter`インスタンスをすることができます。これにより、ユーザーはゼロのコンストラクタをたない`XmlAdapter`をしたり、アダプタにデータをしたりすることができます。

ユーザークラス

のクラスには、とユーザーのがまれます。

```

import java.awt.image.BufferedImage;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;

@XmlRootElement
public class User {

    private String name;
    private BufferedImage image;

    @XmlAttribute
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @XmlJavaTypeAdapter(value=ImageCacheAdapter.class)
    @XmlAttribute
    public BufferedImage getImage() {
        return image;
    }

    public void setImage(BufferedImage image) {
        this.image = image;
    }

    public User(String name, BufferedImage image) {
        this.name = name;
        this.image = image;
    }

    public User() {
        this("", null);
    }

}

```

アダプタ

メモリーにイメージを2するのをけるためにそしてデータをもうダウンロードするだけでなく、アダプターはイメージをマップにします。

Java SE 7

なJava 7コードのは、 `getImage` メソッドを

```

public BufferedImage getImage(URL url) {
    BufferedImage image = imageCache.get(url);
    if (image == null) {
        try {
            image = ImageIO.read(url);
        } catch (IOException ex) {
            Logger.getLogger(ImageCacheAdapter.class.getName()).log(Level.SEVERE, null, ex);
            return null;
        }
    }
    return image;
}

```

```

    }
    imageCache.put(url, image);
    reverseIndex.put(image, url);
}
return image;
}

```

```

import java.awt.image.BufferedImage;
import java.io.IOException;
import java.net.URL;
import java.util.HashMap;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.imageio.ImageIO;
import javax.xml.bind.annotation.adapters.XmlAdapter;

public class ImageCacheAdapter extends XmlAdapter<String, BufferedImage> {

    private final Map<URL, BufferedImage> imageCache = new HashMap<>();
    private final Map<BufferedImage, URL> reverseIndex = new HashMap<>();

    public BufferedImage getImage(URL url) {
        // using a single lookup using Java 8 methods
        return imageCache.computeIfAbsent(url, s -> {
            try {
                BufferedImage img = ImageIO.read(s);
                reverseIndex.put(img, s);
                return img;
            } catch (IOException ex) {
                Logger.getLogger(ImageCacheAdapter.class.getName()).log(Level.SEVERE, null,
ex);
                return null;
            }
        });
    }

    @Override
    public BufferedImage unmarshal(String v) throws Exception {
        return getImage(new URL(v));
    }

    @Override
    public String marshal(BufferedImage v) throws Exception {
        return reverseIndex.get(v).toExternalForm();
    }

}

```

XMLの

の2つのXMLは、*Jon Skeet*とEarth 2にしています。どちらもじょうにえるため、じアバターをし
ます。

```

<?xml version="1.0" encoding="UTF-8"?>

<user name="Jon Skeet"
image="https://www.gravatar.com/avatar/6d8ebb117e8d83d74ea95fbdd0f87e13?s=328&d=identicon&r=PG"

```

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<user name="Jon Skeet (Earth 2)"
```

```
image="https://www.gravatar.com/avatar/6d8ebb117e8d83d74ea95fbdd0f87e13?s=328&d=identicon&r=PG"
```

アダプタの

```
ImageCacheAdapter adapter = new ImageCacheAdapter();

JAXBContext context = JAXBContext.newInstance(User.class);

Unmarshaller unmarshaller = context.createUnmarshaller();

// specify the adapter instance to use for every
// @XmlJavaTypeAdapter(value=ImageCacheAdapter.class)
unmarshaller.setAdapter(ImageCacheAdapter.class, adapter);

User result1 = (User) unmarshaller.unmarshal(Main.class.getResource("user.xml"));

// unmarshal second xml using the same adapter instance
Unmarshaller unmarshaller2 = context.createUnmarshaller();
unmarshaller2.setAdapter(ImageCacheAdapter.class, adapter);
User result2 = (User) unmarshaller2.unmarshal(Main.class.getResource("user2.xml"));

System.out.println(result1.getName());
System.out.println(result2.getName());

// yields true, since image is reused
System.out.println(result1.getImage() == result2.getImage());
```

XMLをなJavaクラスにバインドする。

これは、XMLをなJavaクラスにバインドする `package-info.java` ファイルのです。これは、ネームスペースをしてシリアライズするがあるJavaクラスとじパッケージにするがあります。

```
/**
 * A package containing serializable classes.
 */
@XmlSchema
(
    xmlns =
    {
        @XmlNs(prefix = MySerializableClass.NAMESPACE_PREFIX, namespaceURI =
MySerializableClass.NAMESPACE)
    },
    namespace = MySerializableClass.NAMESPACE,
    elementFormDefault = XmlNsForm.QUALIFIED
)
package com.test.jaxb;

import javax.xml.bind.annotation.XmlNs;
import javax.xml.bind.annotation.XmlNsForm;
import javax.xml.bind.annotation.XmlSchema;
```


XmlAdapterをしてをトリミングする

```
package com.example.xml.adapters;

import javax.xml.bind.annotation.adapters.XmlAdapter;

public class StringTrimAdapter extends XmlAdapter<String, String> {
    @Override
    public String unmarshal(String v) throws Exception {
        if (v == null)
            return null;
        return v.trim();
    }

    @Override
    public String marshal(String v) throws Exception {
        if (v == null)
            return null;
        return v.trim();
    }
}
```

そして、package-info.javaにのをします。

```
@XmlJavaTypeAdapter(value = com.example.xml.adapters.StringTrimAdapter.class, type =
String.class)
package com.example.xml.jaxb.bindings; // Packge where you intend to apply trimming filter

import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;
```

オンラインでJAXBをむ <https://riptutorial.com/ja/java/topic/147/jaxb>

46: JAXP APIをしたXML

XMLは、XMLを、「ノード」、「」、「」、「」、またはこれらのにするイベントのようなやりやすいをしてするためのものです。

Javaには、[JAXP](#)と呼ばれるXMLのネイティブAPI、またはXMLのJava APIがあります。JAXPとリファレンスは、Java 1.4JAXP v1.1のすべてのJavaリリースにバンドルされてをげています。Java 8には、JAXPバージョン1.6がしています。

APIは、XMLとするさまざまなをします。

- DOMインタフェースDocument Object Modelは、
- SAXインタフェースSimple API for XML
- StAXインタフェースXMLストリーミングAPI

DOM インタフェースの

DOMインタフェースは、XMLをする[W3C DOM](#)のをすることをしています。JAXPのさまざまなバージョンでは、さまざまなDOMレベルのレベル3までがサポートされています。

ドキュメントオブジェクトモデルインタフェースのでは、XMLドキュメントはツリーとしてされ、「ドキュメント」からまります。APIのがある[Node](#)タイプ、それからすることをにするNodeではないすべて、が、またはをNode Sはをつことができ、例えば、[Text](#)ノードは、ツリーのもであり、をつことはない。XMLタグは[Element](#)としてされ、にのメソッドでNodeをします。

DOMインタフェースは、XMLドキュメントをツリーとして「1」し、されたツリーノードの、コピーなどのなをし、にそのシリアルディスクにすの。しかし、これはががかかりますツリーはメモリにしているため、DOMツリーはなXMLにはずしもではありません。さらに、ツリーのは、にXMLのすべてののにがないに、XMLコンテンツをするのでありとありません。

SAX インタフェースの

SAX APIは、XMLドキュメントをするイベントのAPIです。このモデルでは、XMLドキュメントのコンポーネントはイベントとしてされますたとえば、「タグがかれた」、「タグがじられた」、「テキストノードにしました」、「コメントがしました」など。..

SAX APIは、「プッシュ・パース」アプローチをします。このアプローチでは、SAX [Parser](#)がXMLドキュメントのをし、デリゲート [ContentHandler](#) のメソッドをびしてXMLドキュメントのイベントをします。、パーサをくことはありませんが、XMLドキュメントからなをすべてするハンドラがされています。

SAXインタフェースは、パーサーレベルたとえば、のコンテキスト、でのデータのみをするこ

とによってDOMインターフェイスのを使します。したがって、`ContentHandler`によってされるのみメモリにされる。このようなアプローチでは、「`XML`にる」がないというトレードオフがあります。DOMでは`Node`がにることができますが、SAXにはそのようなありません。

StAX インターフェイスの

StAX APIは、SAX APIつまりイベントとにXMLにたアプローチをしています。StAXはプル・パーサーSAXはプッシュ・パーサーです。SAXでは、`Parser`はされ、`ContentHandler`コールバックをします。Staxでは、パーサをびし、のXML "イベント"をするかどうかをします。

APIは`XMLStreamReader` または `XMLEventReader` でまります。これは、がイテレータースタイルの `nextEvent()` ことができるゲートウェイです。

Examples

DOM APIをしてドキュメントをおよびナビゲートする

のをしてください。

```
<?xml version='1.0' encoding='UTF-8' ?>
<library>
  <book id='1'>Effective Java</book>
  <book id='2'>Java Concurrency In Practice</book>
</library>
```

のコードをして、`String` からDOMツリーをできます。

```
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.InputSource;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import java.io.StringReader;

public class DOMDemo {

    public static void main(String[] args) throws Exception {
        String xmlDocument = "<?xml version='1.0' encoding='UTF-8' ?>"
            + "<library>"
            + "<book id='1'>Effective Java</book>"
            + "<book id='2'>Java Concurrency In Practice</book>"
            + "</library>";

        DocumentBuilderFactory documentBuilderFactory = DocumentBuilderFactory.newInstance();
        // This is useless here, because the XML does not have namespaces, but this option is
        // usefull to know in cas
        documentBuilderFactory.setNamespaceAware(true);
        DocumentBuilder documentBuilder = documentBuilderFactory.newDocumentBuilder();
        // There are various options here, to read from an InputStream, from a file, ...
```

```

Document document = documentBuilder.parse(new InputSource(new StringReader(xmlDocument)));

// Root of the document
System.out.println("Root of the XML Document: " +
document.getDocumentElement().getLocalName());

// Iterate the contents
NodeList firstLevelChildren = document.getDocumentElement().getChildNodes();
for (int i = 0; i < firstLevelChildren.getLength(); i++) {
    Node item = firstLevelChildren.item(i);
    System.out.println("First level child found, XML tag name is: " +
item.getLocalName());
    System.out.println("\tid attribute of this tag is : " +
item.getAttributes().getNamedItem("id").getTextContent());
}

// Another way would have been
NodeList allBooks = document.getDocumentElement().getElementsByTagName("book");
}
}

```

コードはのようになります。

```

Root of the XML Document: library
First level child found, XML tag name is: book
id attribute of this tag is : 1
First level child found, XML tag name is: book
id attribute of this tag is : 2

```

StAX APIをしたの

のをしてください。

```

<?xml version='1.0' encoding='UTF-8' ?>
<library>
  <book id='1'>Effective Java</book>
  <book id='2'>Java Concurrency In Practice</book>
  <notABook id='3'>This is not a book element</notABook>
</library>

```

のコードをして、それをし、IDごとにタイトルのをすることができます。

```

import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamConstants;
import javax.xml.stream.XMLStreamReader;
import java.io.StringReader;
import java.util.HashMap;
import java.util.Map;

public class StaxDemo {

    public static void main(String[] args) throws Exception {
        String xmlDocument = "<?xml version='1.0' encoding='UTF-8' ?>"
            + "<library>"
            + "<book id='1'>Effective Java</book>"
            + "<book id='2'>Java Concurrency In Practice</book>"

```

```

        + "<notABook id='3'>This is not a book element </notABook>"
        + "</library>";

XMLInputFactory xmlInputFactory = XMLInputFactory.newFactory();
// Various flavors are possible, e.g. from an InputStream, a Source, ...
XMLStreamReader xmlStreamReader = xmlInputFactory.createXMLStreamReader(new
StringReader(xmlDocument));

Map<Integer, String> bookTitlesById = new HashMap<>();

// We go through each event using a loop
while (xmlStreamReader.hasNext()) {
    switch (xmlStreamReader.getEventType()) {
        case XMLStreamConstants.START_ELEMENT:
            System.out.println("Found start of element: " +
xmlStreamReader.getLocalName());
            // Check if we are at the start of a <book> element
            if ("book".equals(xmlStreamReader.getLocalName())) {
                int bookId = Integer.parseInt(xmlStreamReader.getAttributeValue("",
"id"));

                String bookTitle = xmlStreamReader.getElementText();
                bookTitlesById.put(bookId, bookTitle);
            }
            break;
        // A bunch of other things are possible : comments, processing instructions,
Whitespace...
        default:
            break;
    }
    xmlStreamReader.next();
}

System.out.println(bookTitlesById);
}

```

これは、

```

Found start of element: library
Found start of element: book
Found start of element: book
Found start of element: notABook
{1=Effective Java, 2=Java Concurrency In Practice}

```

このサンプルでは、いくつかのことをにすることがあります。

1. `xmlStreamReader.getAttributeValue` は、パーサが `START_ELEMENT` にあることをにしたために `START_ELEMENT` ます。の `ATTRIBUTES` をくでは、パーサーは `IllegalStateException` をスローしなければなりません。はのにしかれないからです。
2. `xmlStreamReader.getTextContent()` じですが、これは `START_ELEMENT` あり、このドキュメントでは `<book>` にテキストのノードがないことがわかっているためします。

よりなく、れになったなどは、パーサーをサブメソッドやの `BookParser` に「する」 `BookParser` クラスやメソッドをつなど、すべてのをすることをお `BookParser` ますのXMLタグの `START_ELEMENT` から `END_ELEMENT` まで

Stackオブジェクトをして、なデータをツリーのにすることもできます。

オンラインでJAXP APIをしたXMLをむ <https://riptutorial.com/ja/java/topic/3943/jaxp-api>をしたxml

47: JAX-WS

Examples

をしてJAX-WSびしをうは、ほとんどありません。

に、 `Service` がサービスクラスなので、 `Port` がアクセスするサービスポートであるをします。

```
Service s = new Service();
Port port = s.getPort();

BindingProvider prov = (BindingProvider)port;
prov.getRequestContext().put(BindingProvider.USERNAME_PROPERTY, "myusername");
prov.getRequestContext().put(BindingProvider.PASSWORD_PROPERTY, "mypassword");

port.call();
```

オンラインでJAX-WSをむ <https://riptutorial.com/ja/java/topic/4105/jax-ws>

48: JMX

き

JMXテクノロジーは、デバイス、アプリケーション、およびサービスネットワークをおよびするためのWebベースのモジュラおよびダイナミックソリューションをするためのツールをします。、これは、レガシーシステムの、しいおよびソリューションの、およびのシステムへののにしています。

Examples

プラットフォームMBeanサーバのな

たとえば、しいユーザーをしてメッセージをするサーバーがあるとします。そして、私たちはこのサーバーをし、そのサーバーのをしたいとえています。

まず、とのとのインターフェースがです

```
public interface UserCounterMBean {
    long getSleepTime();

    void setSleepTime(long sleepTime);

    int getUserCount();

    void setUserCount(int userCount);

    String getGreetingString();

    void setGreetingString(String greetingString);

    void stop();
}
```

そしてそれがどのようにしているのか、それがどのようにをえているのかをていくな

```
public class UserCounter implements UserCounterMBean, Runnable {
    private AtomicLong sleepTime = new AtomicLong(10000);
    private AtomicInteger userCount = new AtomicInteger(0);
    private AtomicReference<String> greetingString = new AtomicReference<>("welcome");
    private AtomicBoolean interrupted = new AtomicBoolean(false);

    @Override
    public long getSleepTime() {
        return sleepTime.get();
    }

    @Override
    public void setSleepTime(long sleepTime) {
        this.sleepTime.set(sleepTime);
    }
}
```



```

    }

    @Override
    public int getUserCount() {
        return userCount.get();
    }

    @Override
    public void setUserCount(int userCount) {
        this.userCount.set(userCount);
    }

    @Override
    public String getGreetingString() {
        return greetingString.get();
    }

    @Override
    public void setGreetingString(String greetingString) {
        this.greetingString.set(greetingString);
    }

    @Override
    public void stop() {
        this.interrupted.set(true);
    }

    @Override
    public void run() {
        while (!interrupted.get()) {
            try {
                System.out.printf("User %d, %s%n", userCount.incrementAndGet(),
greetingString.get());
                Thread.sleep(sleepTime.get());
            } catch (InterruptedException ignored) {
            }
        }
    }
}

```

ローカルまたはリモートのなのために、たちのMBeanをするがあります

```

import javax.management.InstanceAlreadyExistsException;
import javax.management.MBeanRegistrationException;
import javax.management.MBeanServer;
import javax.management.MalformedObjectNameException;
import javax.management.NotCompliantMBeanException;
import javax.management.ObjectName;
import java.lang.management.ManagementFactory;

public class Main {
    public static void main(String[] args) throws MalformedObjectNameException,
NotCompliantMBeanException, InstanceAlreadyExistsException, MBeanRegistrationException,
InterruptedException {
        final UserCounter userCounter = new UserCounter();
        final MBeanServer mBeanServer = ManagementFactory.getPlatformMBeanServer();
        final ObjectName objectName = new ObjectName("ServerManager:type=UserCounter");
        mBeanServer.registerMBean(userCounter, objectName);

        final Thread thread = new Thread(userCounter);
    }
}

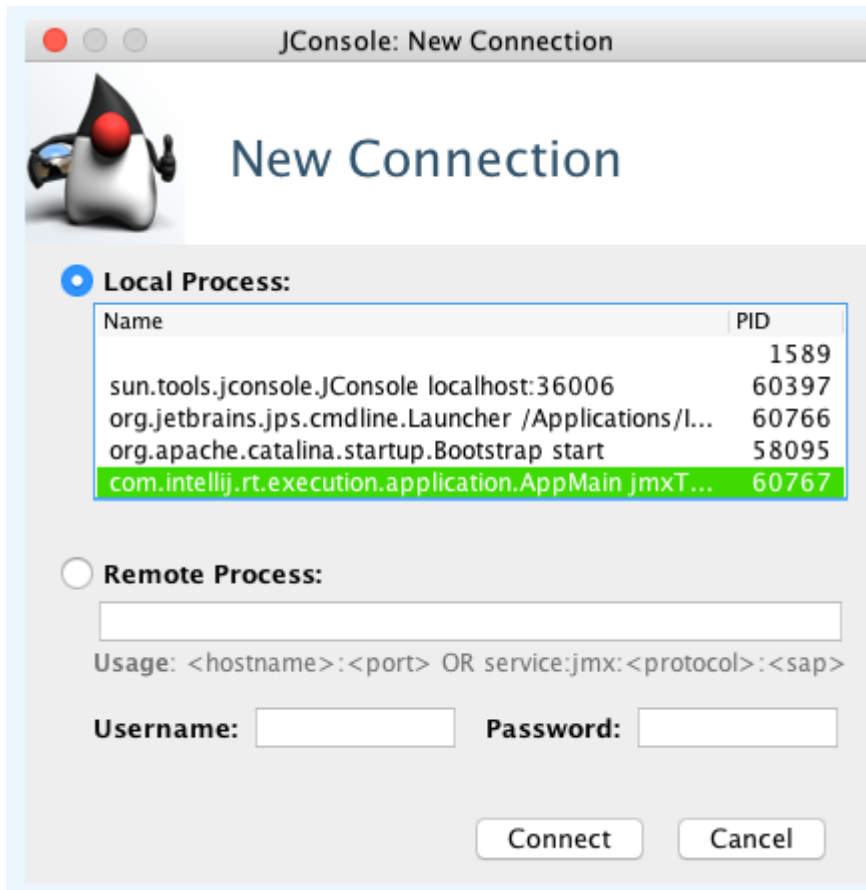
```

```

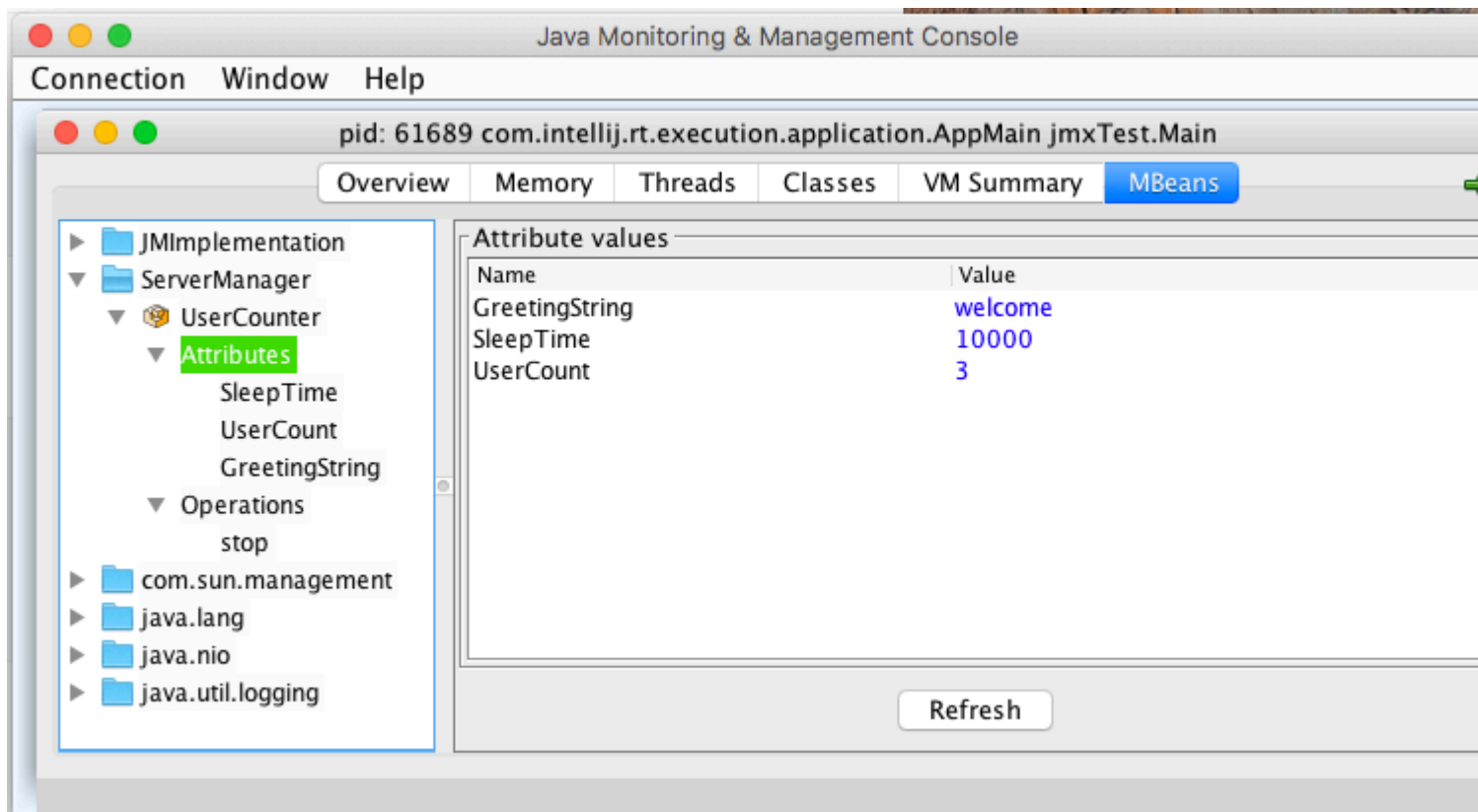
        thread.start();
        thread.join();
    }
}

```

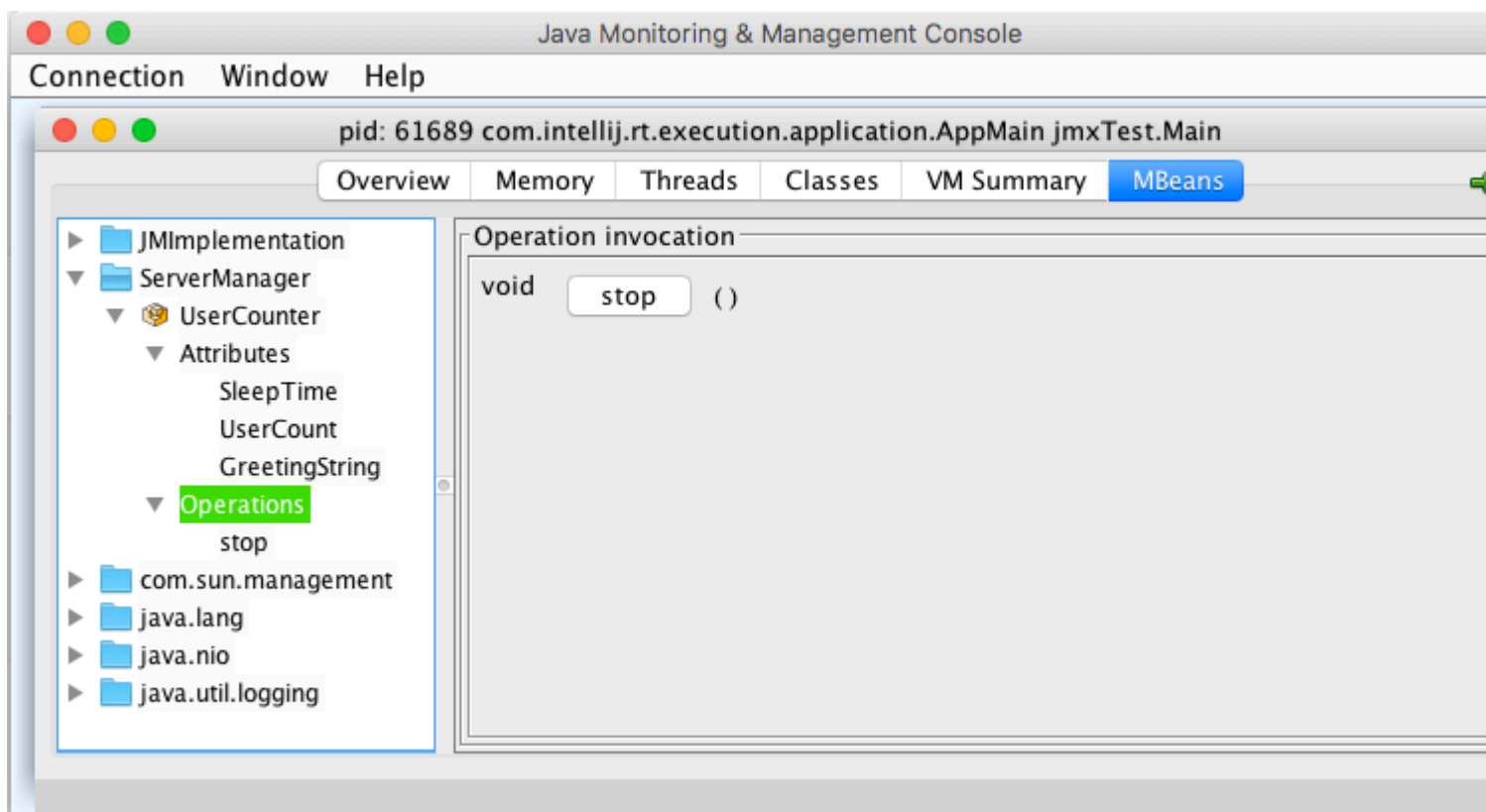
その、アプリケーションをして、`$JAVA_HOME/bin`ディレクトリにあるjConsoleでアプリケーションにできます。まず、アプリケーションをしてローカルjavaプロセスをつけるがあります



MBeanタブにりえ、メインクラスでしたMBeanをObjectName のではServerManager としてします。Attributes セクションでは、Attributes をることができます。getメソッドのみをした、はみりですがきみではありません。getメソッドとsetメソッドのをした、はみきになります。



されたメソッドは、Operationsセクションでびすことができます。



リモートをできるようにするには、のようなJVMパラメータがです。

```
-Dcom.sun.management.jmxremote=true //true by default
-Dcom.sun.management.jmxremote.port=36006
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
```

これらのパラメータは、[JMXガイドの2にあります](#)。その、jConsoleの `jconsole host:port` をして jConsoleでアプリケーションにするか、jConsoleのGUIで `host:port` または `service:jmx:rmi:///jndi/rmi://hostName:portNum/jmxrmi` をしてアプリケーションにできます。

につリンク

- [JMXガイド](#)
- [JMXベストプラクティス](#)

オンラインでJMXをむ <https://riptutorial.com/ja/java/topic/9278/jmx>

49: JNDI

Examples

JNDIによるRMI

これは、JNDIがRMIでどのようにするかをしています。それには2つのがあります。

- RMIレジストリへのbind / unbind / rebind APIをサーバにする
- RMIレジストリへの/リストAPIをクライアントにする。

RMIレジストリは、JNDIではなくRMIのです。

これをにするために、 `java.rmi.registry.CreateRegistry()` をしてRMIレジストリをします。

1. Server.javaJNDIサーバー

```
package com.neohope.jndi.test;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.io.IOException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.util.Hashtable;

/**
 * JNDI Server
 * 1.create a registry on port 1234
 * 2.bind JNDI
 * 3.wait for connection
 * 4.clean up and end
 */
public class Server {
    private static Registry registry;
    private static InitialContext ctx;

    public static void initJNDI() {
        try {
            registry = LocateRegistry.createRegistry(1234);
            final Hashtable jndiProperties = new Hashtable();
            jndiProperties.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.rmi.registry.RegistryContextFactory");
            jndiProperties.put(Context.PROVIDER_URL, "rmi://localhost:1234");
            ctx = new InitialContext(jndiProperties);
        } catch (NamingException e) {
            e.printStackTrace();
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }

    public static void bindJNDI(String name, Object obj) throws NamingException {
```

```

        ctx.bind(name, obj);
    }

    public static void unbindJNDI(String name) throws NamingException {
        ctx.unbind(name);
    }

    public static void unInitJNDI() throws NamingException {
        ctx.close();
    }

    public static void main(String[] args) throws NamingException, IOException {
        initJNDI();
        NMessage msg = new NMessage("Just A Message");
        bindJNDI("/neohope/jndi/test01", msg);
        System.in.read();
        unbindJNDI("/neohope/jndi/test01");
        unInitJNDI();
    }
}

```

2. Client.javaJNDIクライアント

```

package com.neohope.jndi.test;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Hashtable;

/**
 * 1.init context
 * 2.lookup registry for the service
 * 3.use the service
 * 4.end
 */
public class Client {
    public static void main(String[] args) throws NamingException {
        final Hashtable jndiProperties = new Hashtable();
        jndiProperties.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.rmi.registry.RegistryContextFactory");
        jndiProperties.put(Context.PROVIDER_URL, "rmi://localhost:1234");

        InitialContext ctx = new InitialContext(jndiProperties);
        NMessage msg = (NeoMessage) ctx.lookup("/neohope/jndi/test01");
        System.out.println(msg.message);
        ctx.close();
    }
}

```

3. NMessage.javaRMIサーバークラス

```

package com.neohope.jndi.test;

import java.io.Serializable;
import java.rmi.Remote;

/**

```

```

* NMessage
* RMI server class
* must implements Remote and Serializable
*/
public class NMessage implements Remote, Serializable {
    public String message = "";

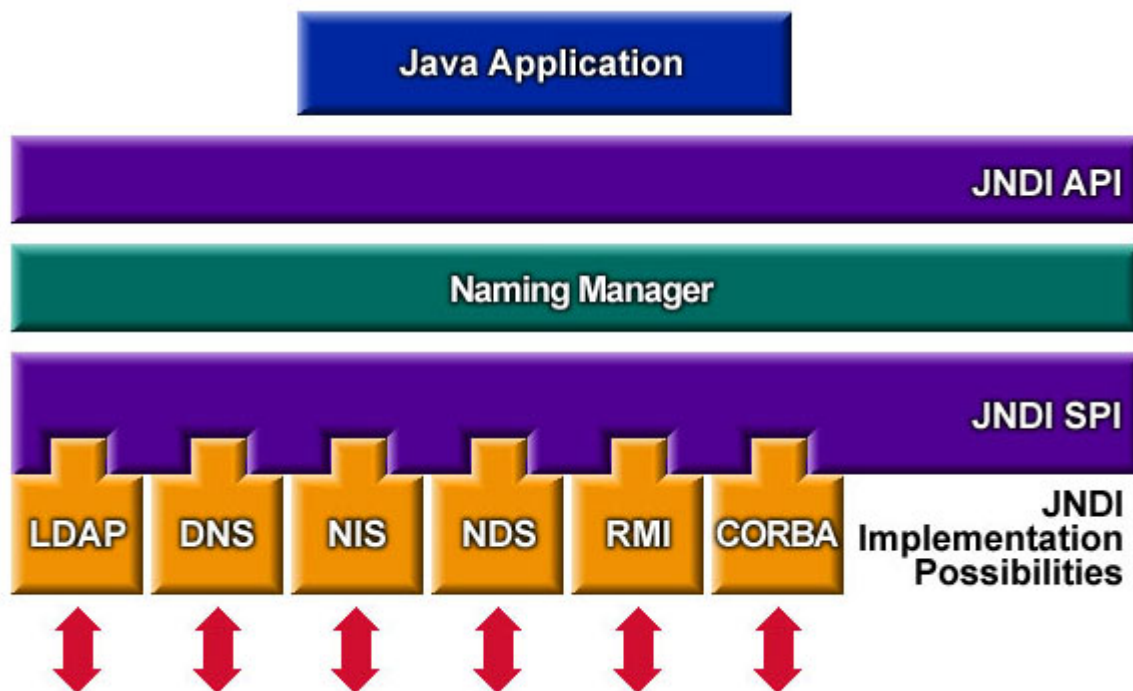
    public NMessage(String message)
    {
        this.message = message;
    }
}

```

exampleの

1. サーバーのと
2. クライアントをビルドしてする

する



Java Naming and Directory Interface JNDIは、ディレクトリサービスのJava APIであり、Java ソフトウェアクライアントがをしてデータとオブジェクトをおよびできるようにします。のネーミングまたはディレクトリサービスのとはしてされています。

JNDIアーキテクチャは、**API Application Programming Interface**と**SPI Service Provider Interface**でされています。JavaアプリケーションはこのAPIをして、さまざまなネーミングおよびディレクトリサービスにアクセスします。SPIをすると、さまざまなネーミングサービスとディレクトリサービスをにプラグインすることができ、JNDIテクノロジーのAPIをしてJavaアプリケーションがサービスにアクセスできるようになります。

ののように、JNDIはLDAP、DNS、NIS、NDS、RMI、CORBAをサポートしています。もちろん

、あなたはそれを行うことができます。

い

ここでは、Java RMIはJNDI APIをしてネットワークのオブジェクトをルックアップします。オブジェクトをルックアップするには、少なくとも2つのが必要です。

- オブジェクトをつける

RMIレジストリはバインディングをし、オブジェクトのをします。

- オブジェクトの

オブジェクトのはですかこれはであり、Nameインタフェースをするオブジェクトでもあります。

ステップバイステップ

1. まず、のバインドをするレジストリがです。ここでは、 `java.rmi.registry.LocateRegistry` をします。

```
//This will start a registry on localhost, port 1234
registry = LocateRegistry.createRegistry(1234);
```

2. クライアントとサーバーのにコンテキストがです。サーバーは、コンテキストをしてとオブジェクトをバインドします。クライアントはコンテキストをしてをし、オブジェクトをします。

```
//We use com.sun.jndi.rmi.registry.RegistryContextFactory as the InitialContextFactory
final Hashtable jndiProperties = new Hashtable();
jndiProperties.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.rmi.registry.RegistryContextFactory");
//the registry url is "rmi://localhost:1234"
jndiProperties.put(Context.PROVIDER_URL, "rmi://localhost:1234");
InitialContext ctx = new InitialContext(jndiProperties);
```

3. サーバーは、とオブジェクトをバインドします。

```
//The jndi name is "/neohope/jndi/test01"
bindJNDI("/neohope/jndi/test01", msg);
```

4. クライアントはオブジェクトを `"/neohope/jndi/test01"` というでルックアップします。

```
//look up the object by name "java:com/neohope/jndi/test01"
NeoMessage msg = (NeoMessage) ctx.lookup("/neohope/jndi/test01");
```

5. これで、クライアントはオブジェクトをできます
6. サーバーがしたら、クリーンアップするがあります。


```
ctx.unbind("/neohope/jndi/test01");  
ctx.close();
```

オンラインでJNDIをむ <https://riptutorial.com/ja/java/topic/5720/jndi>

50: JShell

き

JShellは、JDK 9でされたJavaのインタラクティブなREPLです。は、をしたり、クラスをテストしたり、Javaでしたりすることができます。jdk 9のアクセスは<http://jdk.java.net/9/>からできます。

- `$ jshell` - JShell REPLをする
- `jshell> / <command>` - えられたJShellコマンドをする
- `jshell> / exit` - JShellをする
- `jshell> / help` - JShellコマンドのリストをする
- `jshell> <java_expression>` - されたJavaをするセミコロンは
- `jshell> / vars OR / methods OR / types` - 、メソッド、またはクラスのリストをそれぞれしてください。
- `jshell> / open <file>` - シェルへのかしてファイルをみむ
- `jshell> / edit <identifier>` - セットエディタでスニペットをする
- `jshell> / set editor <command>` - / editをしてスニペットのかするコマンドをする
- `jshell> / drop <identifier>` - スニペットをする
- `jshell> / reset` - JVMをリセットしてすべてのスニペットをする

JShellにはJava 9 JDKがです。JDKは20173 jdk9.java.netからアクセススナップショットとしてダウンロードできます。 `jshell` コマンドをしようとするとき `Unable to locate an executable` が `Unable to locate an executable` というエラーでまり、 `JAVA_HOME` がしくされていることをし `JAVA_HOME` 。

デフォルト

のパッケージは、JShellのかにインポートされます。

```
import java.io.*
import java.math.*
import java.net.*
import java.nio.file.*
import java.util.*
import java.util.concurrent.*
import java.util.function.*
import java.util.prefs.*
import java.util.regex.*
import java.util.stream.*
```

Examples

JShellのかと

JShellの

JShellをするに、`JAVA_HOME`がJDK 9インストールをしていることをしてください。JShellをするには、のコマンドをします。

```
$ jshell
```

すべてうまくいけば、`jshell>`プロンプトがされます。

JShellの

JShellをするには、JShellプロンプトからのコマンドをします。

```
jshell> /exit
```

JShellでは、セミコロンのにかかわらず、Javaをできます。これらは、なやからよりなものにふことがあります。

```
jshell> 4+2
jshell> System.out.printf("I am %d years old.\n", 421)
```

ループともありません。

```
jshell> for (int i = 0; i<3; i++) {
...> System.out.println(i);
...> }
```

ブロックのはセミコロンでなければならないことにすることがです。

JShellでローカルをすることができます

```
jshell> String s = "hi"
jshell> int i = s.length
```

はさまざまなでできます。これはJShellでにです。

```
jshell> String var = "hi"
jshell> int var = 3
```

のリストをするには、JShellプロンプトで`/vars`とします。

メソッドとクラス

JShellでメソッドとクラスをできます。

```
jshell> void speak() {
...> System.out.println("hello");
...> }

jshell> class MyClass {
...> void doNothing() {}
...> }
```

アクセスはありません。のブロックとに、メソッドのにはセミコロンがです。のとに、メソッドやクラスをすることもです。メソッドまたはクラスのリストをするには、JShellプロンプトで `/methods` または `/types` をそれぞれします。

スニペットの

JShellでされるコードのはスニペットまたはソースエントリです。ローカルをするか、ローカルメソッドまたはクラスをするたびに、そのが `/メソッド/クラス` のであるスニペットをします。いつでも `/edit` コマンドでしたスニペットを `/edit` 。たとえば、が `Foo` というクラスをメソッド、 `bar` したとしましょう

```
jshell> class Foo {
...> void bar() {
...> }
...> }
```

さて、はののをめることをみます。クラスをリライトするのではなく、することができます

```
jshell> /edit Foo
```

デフォルトでは、スイングエディタがなもなをポップアップします。ただし、JShellでするエディタをすることはできます。

```
jshell> /set editor emacs
jshell> /set editor vi
jshell> /set editor nano
jshell> /set editor -default
```

スニペットのしいバージョンにエラーがまれているは、されないことがあります。に、スニペットは、の/がにしいにのみされます。はしません。

```
jshell> String st = String 3
//error omitted
jshell> /edit st
| No such snippet: st
```

しかし、スニペットがコンパイルされ、コンパイルにエラーがしてもできます。たとえば、のようします。

```
jshell> int i = "hello"
//error omitted
```

```
jshell> /edit i
```

に、 `/drop` コマンドをしてスニペットをすることができます。

```
jshell> int i = 13
jshell> /drop i
jshell> System.out.println(i)
| Error:
| cannot find symbol
|   symbol:   variable i
|   System.out.println(i)
|
```

すべてのスニペットをし、それによってJVMのを `\reset` するには、 `\reset` します。

```
jshell> int i = 2

jshell> String s = "hi"

jshell> /reset
| Resetting state.

jshell> i
| Error:
| cannot find symbol
|   symbol:   variable i
|   i
|   ^

jshell> s
| Error:
| cannot find symbol
|   symbol:   variable s
|   s
|   ^
```

オンラインでJShellをむ <https://riptutorial.com/ja/java/topic/9511/jshell>

51: JVM ツールインタフェース

JVM TM ツールインタフェース

バージョン1.2

<http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html>

Examples

オブジェクトからなオブジェクトをするヒープ1.0

```
#include <vector>
#include <string>

#include "agent_util.hpp"
//this file can be found in Java SE Development Kit 8u101 Demos and Samples
//see http://download.oracle.com/otn-pub/java/jdk/8u101-b13-demos/jdk-8u101-windows-x64-
demos.zip
//jdk1.8.0_101.zip!\demo\jvmti\versionCheck\src\agent_util.h

/*
 * Struct used for jvmti->SetTag(object, <pointer to tag>);
 * http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#SetTag
 */
typedef struct Tag
{
    jlong referrer_tag;
    jlong size;
    char* classSignature;
    jint hashCode;
} Tag;

/*
 * Utility function: jlong -> Tag*
 */
static Tag* pointerToTag(jlong tag_ptr)
{
    if (tag_ptr == 0)
    {
        return new Tag();
    }
    return (Tag*)(ptrdiff_t)(void*)tag_ptr;
}

/*
 * Utility function: Tag* -> jlong
 */
static jlong tagToPointer(Tag* tag)
{
    return (jlong)(ptrdiff_t)(void*)tag;
}
```

```

/*
 * Heap 1.0 Callback
 * http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#jvmtiObjectReferenceCallback
 */
static jvmtiIterationControl JNICALL heapObjectReferencesCallback(
    jvmtiObjectReferenceKind reference_kind,
    jlong class_tag,
    jlong size,
    jlong* tag_ptr,
    jlong referrer_tag,
    jint referrer_index,
    void* user_data)
{
    //iterate only over reference field
    if (reference_kind != JVMTI_HEAP_REFERENCE_FIELD)
    {
        return JVMTI_ITERATION_IGNORE;
    }
    auto tag_ptr_list = (std::vector<jlong>*) (ptrdiff_t) (void*)user_data;
    //create and assign tag
    auto t = pointerToTag(*tag_ptr);
    t->referrer_tag = referrer_tag;
    t->size = size;
    *tag_ptr = tagToPointer(t);
    //collect tag
    (*tag_ptr_list).push_back(*tag_ptr);

    return JVMTI_ITERATION_CONTINUE;
}

/*
 * Main function for demonstration of Iterate Over Objects Reachable From Object
 *
 * http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#IterateOverObjectsReachableFromObject
 */
void iterateOverObjectHeapReferences(jvmtiEnv* jvmti, JNIEnv* env, jobject object)
{
    std::vector<jlong> tag_ptr_list;

    auto t = new Tag();
    jvmti->SetTag(object, tagToPointer(t));
    tag_ptr_list.push_back(tagToPointer(t));

    stdout_message("tag list size before call callback:  %d\n", tag_ptr_list.size());
    /*
     * Call Callback for every reachable object reference
     * see
     * http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#IterateOverObjectsReachableFromObject
     */
    jvmti->IterateOverObjectsReachableFromObject(object, &heapObjectReferencesCallback,
    (void*)&tag_ptr_list);
    stdout_message("tag list size after call callback:  %d\n", tag_ptr_list.size());

    if (tag_ptr_list.size() > 0)
    {
        jint found_count = 0;
        jlong* tags = &tag_ptr_list[0];
        jobject* found_objects;
    }
}

```

```

    jlong* found_tags;

    /*
    *    collect all tagged object (via *tag_ptr = pointer to tag )
    *    see
    http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#GetObjectsWithTags
    */
    jvmti->GetObjectsWithTags(tag_ptr_list.size(), tags, &found_count, &found_objects,
    &found_tags);
    stdout_message("found %d objects\n", found_count);

    for (auto i = 0; i < found_count; ++i)
    {
        jobject found_object = found_objects[i];

        char* classSignature;
        jclass found_object_class = env->GetObjectClass(found_object);
        /*
        *    Get string representation of found_object_class
        *    see
    http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#GetClassSignature
        */
        jvmti->GetClassSignature(found_object_class, &classSignature, nullptr);

        jint hashCode;
        /*
        *    Getting hash code for found_object
        *    see
    http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#GetObjectHashCode
        */
        jvmti->GetObjectHashCode(found_object, &hashCode);

        //save all it in Tag
        Tag* t = pointerToTag(found_tags[i]);
        t->classSignature = classSignature;
        t->hashCode = hashCode;
    }

    //print all saved information
    for (auto i = 0; i < found_count; ++i)
    {
        auto t = pointerToTag(found_tags[i]);
        auto rt = pointerToTag(t->referrer_tag);

        if (t->referrer_tag != 0)
        {
            stdout_message("referrer object %s#%d --> object %s#%d (size: %2d)\n",
                rt->classSignature, rt->hashCode, t->classSignature, t->hashCode, t-
>size);
        }
    }
}
}

```

JVMTIをする

Agent_OnLoadメソッド

```
jvmtiEnv* jvmti;
```



```
/* Get JVMTI environment */
vm->GetEnv(reinterpret_cast<void **>(&jvmti), JVMTI_VERSION);
```

Agent_OnLoad メソッドの

```
/* Callback for JVMTI_EVENT_VM_INIT */
static void JNICALL vm_init(jvmtiEnv* jvmti, JNIEnv* env, jthread thread)
{
    jint runtime_version;
    jvmti->GetVersionNumber(&runtime_version);
    stdout_message("JVMTI Version: %d\n", runtime_verision);
}

/* Agent_OnLoad() is called first, we prepare for a VM_INIT event here. */
JNIEXPORT jint JNICALL
Agent_OnLoad(JavaVM* vm, char* options, void* reserved)
{
    jint rc;
    jvmtiEventCallbacks callbacks;
    jvmtiCapabilities capabilities;
    jvmtiEnv* jvmti;

    /* Get JVMTI environment */
    rc = vm->GetEnv(reinterpret_cast<void **>(&jvmti), JVMTI_VERSION);
    if (rc != JNI_OK)
    {
        return -1;
    }

    /* Immediately after getting the jvmtiEnv* we need to ask for the
     * capabilities this agent will need.
     */
    jvmti->GetCapabilities(&capabilities);
    capabilities.can_tag_objects = 1;
    jvmti->AddCapabilities(&capabilities);

    /* Set callbacks and enable event notifications */
    memset(&callbacks, 0, sizeof(callbacks));
    callbacks.VMInit = &vm_init;

    jvmti->SetEventCallbacks(&callbacks, sizeof(callbacks));
    jvmti->SetEventNotificationMode(JVMTI_ENABLE, JVMTI_EVENT_VM_INIT, nullptr);

    return JNI_OK;
}
```

オンラインでJVMツールインタフェースをむ <https://riptutorial.com/ja/java/topic/3316/jvm`ツールインタフェース>

52: JVM フラグ

のオプションのみをすることをくおめします。

- あなたがあなたのシステムをにしているなら。
- にすると、これらのオプションがシステムのやパフォーマンスにをえるがあることにしてください。

Javaのからされた。

Examples

-XXアグレッシブ

-XXaggressiveは、JVMをでし、できるだけくしたにするためののです。このをするために、JVMはによりくのリソースをします。ただし、にすると、のいがになります。このオプションは、され、メモリをにするアプリケーションだけにすることをおめします。

```
-XXaggressive:<param>
```

<param>	
opt	をにスケジュールし、のリリースではデフォルトとなるしいをにします。
memory	メモリのいワークロードにメモリシステムをし、のメモリリソースをしていスループットをできるようにします。JRockit JVMは、であれば、きなページもします。

-XXallocClearChunks

このオプションをすると、TLAりてにとのTLAをクリアし、のチャンクをプリフェッチできます。、そののものがされている、デフォルトは0またはnullによってなりますです。なで、これらのとをクリアしてヒープのメモリーをし、Javaがまたはできるようにするがあります。オブジェクトがりてられているとき、またはこのオプションをしてしいTLAをしたときにできます。

```
-XXallocClearChunks
```

```
-XXallocClearChunks=<true | false>
```

はbooleanオプションであり、IA64システムではにされています。にはそのはアプリケーションにします。クリアされたチャンクのサイズをするは、このオプションを-XXallocClearChunkSizeとみわせて-XXallocClearChunkSize。このフラグをするがブールをしな、デフォルトはtrueです。

-XXAllocClearChunkSize

`-XXAllocClearChunkSize`と`-XXAllocClearChunkSize`にする`-XXAllocClearChunkSize`、このオプションは、されるチャンクのサイズをします。このフラグがされていてもがされていない、デフォルトは512バイトです。

```
-XXAllocClearChunks -XXAllocClearChunkSize=<size> [k|K] [m|M] [g|G]
```

-XXcallProfiling

このオプションをすると、コードのにコールプロファイリングをできます。プロファイリングでは、アプリケーションにのながされ、くの、パフォーマンスがします。これは、JVMがこれらのをできるためです。

このオプションは、JRockit JVM R27.3.0のバージョンでサポートされています。のバージョンではデフォルトになることがあります。

```
java -XXcallProfiling myApp
```

このオプションはデフォルトでになっています。それをにするがあります。

-XXdisableFatSpin

このオプションは、Javaのファットロックスピンコードをにし、ファットロックのをブロックするスレッドがスリープになるようにします。

Javaのオブジェクトは、スレッドがそのオブジェクトのブロックになるとすぐにロックになります。すべてのロックは、ロックスレッドによってされるまでされますつまり、ロックされたままです。ロックがにくされないは、「ファットロック」にらませることができます。「スピニング」は、のロックをにチェックしてロックがしているかどうかをするスレッドがタイトなループでチェックされます。ファットロックにするスピニングは、にはですが、によってはでパフォーマンスにするがあります。 `-XXdisableFatSpin`すると、ったロックにしてスピンをオフにし、なパフォーマンスのヒットをなくすことができます。

```
-XXdisableFatSpin
```

-XXdisableGCHuristics

このオプションはガベージコレクタのをにします。コンパクションヒューリスティックおよびナーサリサイズヒューリスティックは、このオプションのをけません。では、ガベージコレクションヒューリスティックがになっています。

```
-XXdisableFatSpin
```

-XXdumpSize

このオプションをすると、ダンプファイルがされ、そのファイルのサイズ、をできます。

```
-XXdumpsize:<size>
```

<サイズ>	
none	ダンプファイルをしません。
small	Windowsでは、さなダンプ・ファイルがされますLinuxではフル・コア・ダンプがされます。さなダンプには、トレースをむスレッドスタックだけがまれ、そのはほとんどありません。これはJRockit JVM 8.1のサービスパック1と2、およびサービスパック3の7.0のデフォルトです。
normal	すべてのプラットフォームでのダンプをさせます。このダンプ・ファイルには、Javaヒープをくすべてのメモリーがまれます。これは、JRockit JVM 1.4.2のデフォルトです。
large	Javaヒープをむメモリのすべてのものをみます。このオプションは、 <code>-XXdumpSize</code> とにし <code>-XXdumpFullState</code> 。

-XXexitOnOutOfMemory

このオプションをすると、メモリエラーがにしたときにJRockit JVMがします。メモリエラーをするのではなく、JRockit JVMのインスタンスをしたいにできます。にこのコマンドをすると、メモリエラーがしたときににJRockit JVMをさせることができます。

```
-XXexitOnOutOfMemory
```

オンラインでJVMフラグをむ [https://riptutorial.com/ja/java/topic/2500/jvm フラグ](https://riptutorial.com/ja/java/topic/2500/jvm%20フラグ)

53: log4j / log4j2

き

[Apache Log4j](#)はJavaベースのロギング・ユーティリティであり、いくつかのJavaロギング・フレームワークの1つです。このトピックでは、Log4jをJavaでセットアップしてするを、なすべてのについてのなとともにします。

- `Logger.debug "text to log";` //デバッグをする
- `Logger.info "text to log";` //をする
- `Logger.error "text to log";` //ロギングエラー
- `Logger.warn "text to log";` //のロギング
- `Logger.trace "text to log";` //トレースをする
- `Logger.fatal "text to log";` //なエラーをする
- パラメータロギングをしたLog4j2の
- `Logger.debug "デバッグparams {} {} {}"、 param1、 param2、 param3;` //パラメータをしてデバッグをする
- `Logger.info "Info params {} {} {}"、 param1、 param2、 param3;` //パラメータきのロギング
- `Logger.error "エラーparams {} {} {}"、 param1、 param2、 param3;` //パラメータのロギングエラー
- `Logger.warn "params {} {} {}"、 param1、 param2、 param3をします。` //パラメータでをする
- `Logger.trace "Trace params {} {} {"、 " param1、 param2、 param3;` //パラメータをしてトレースをする
- `Logger.fatal "なparams {} {} {}"、 param1、 param2、 param3;` //パラメータでなロギング
- `Logger.error "Caught Exception"、 ex;` //メッセージとスタックトレースをうのロギングにされます

Log4j 1のがわった

201585、Logging ServicesプロジェクトはLog4j 1.xのがしたことをしました。のについては、[Apache Blog](#)をしてください。 **Log4j 1**のユーザーは、[Apache Log4j 2](#)にアップグレードすることをおめします。

[http : //logging.apache.org/log4j/1.2/](http://logging.apache.org/log4j/1.2/)

Examples

Log4jの

のバージョンlog4j2

Mavenの

POM.xml ファイルにのをします。

```
<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.6.2</version>
  </dependency>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.6.2</version>
  </dependency>
</dependencies>
```

Ivyをう

```
<dependencies>
  <dependency org="org.apache.logging.log4j" name="log4j-api" rev="2.6.2" />
  <dependency org="org.apache.logging.log4j" name="log4j-core" rev="2.6.2" />
</dependencies>
```

Gradleをする

```
dependencies {
    compile group: 'org.apache.logging.log4j', name: 'log4j-api', version: '2.6.2'
    compile group: 'org.apache.logging.log4j', name: 'log4j-core', version: '2.6.2'
}
```

log4j 1.xの

Log4j 1.xはEOLEnd-Of-Lifeにしました「」を。

Mavenの

このをPOM.xml ファイルにします。

```
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.17</version>
</dependency>
```

Ivyをう

```
<dependency org="log4j" name="log4j" rev="1.2.17"/>
```

Usign Gradle

```
compile group: 'log4j', name: 'log4j', version: '1.2.17'
```

Buildrの

```
'log4j:log4j:jar:1.2.17'
```

でパスビルドをする

Log4j [ウェブサイトプロジェクト](#)からダウンロードする

Log4jをJavaコードでする

に `final static logger` オブジェクトをするがあります。

```
final static Logger logger = Logger.getLogger(classname.class);
```

に、ロギングメソッドをびします。

```
//logs an error message
logger.info("Information about some param: " + parameter); // Note that this line could throw
a NullPointerException!

//in order to improve performance, it is advised to use the `isXXXEnabled()` Methods
if( logger.isInfoEnabled() ){
    logger.info("Information about some param: " + parameter);
}

// In log4j2 parameter substitution is preferable due to readability and performance
// The parameter substitution only takes place if info level is active which obsoletes the use
of isXXXEnabled().
logger.info("Information about some param: {}" , parameter);

//logs an exception
logger.error("Information about some error: ", exception);
```

プロパティファイルの

Log4jをすると、にデータをコンソールとファイルにすることができます。 `log4j.properties` ファイルをし、こののにれます。

```
# Root logger option
log4j.rootLogger=DEBUG, stdout, file

# Redirect log messages to console
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n

# Redirect log messages to a log file, support file rolling.
```

```
log4j.appender.file=org.apache.log4j.RollingFileAppender
log4j.appender.file.File=C:\\log4j-application.log
log4j.appender.file.MaxFileSize=5MB
log4j.appender.file.MaxBackupIndex=10
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n
```

Mavenをしているは、このプロパティファイルをのパスにねます。

```
/ProjectFolder/src/java/resources
```

なlog4j2.xmlファイル

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
  <Appenders>
    <Console name="STDOUT" target="SYSTEM_OUT">
      <PatternLayout pattern="%d %-5p [%t] %C{2} %m%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Root level="debug">
      <AppenderRef ref="STDOUT"/>
    </Root>
  </Loggers>
</Configuration>
```

これは、コンソールアペンダーとルートロガーをつなlog4j2.xmlです。パターンレイアウトは、ステートメントのロギングにどのパターンをすべきかをします。

log4j2.xmlのロードをデバッグするには、status = <WARN | DEBUG | ERROR | FATAL | TRACE | INFO>をlog4j2.xmlのタグにします。

モニターをして、されたのにをロードすることもできます。モニターは、タグにのようになりますmonitorInterval = 30。これは、が30ごとにロードされることをします。

log4j 1.xから2.xへの

プロジェクトのlog4j 1.xからlog4j 2.xにするは、のlog4j 1.xのをすべてし、のをします。

Log4j 1.x API Bridge

Mavenビルド

```
<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-1.2-api</artifactId>
    <version>2.6.2</version>
  </dependency>
</dependencies>
```

アイビービルド


```
<dependencies>
  <dependency org="org.apache.logging.log4j" name="log4j-1.2-api" rev="2.6.2" />
</dependencies>
```

Gradle Build

```
dependencies {
  compile group: 'org.apache.logging.log4j', name: 'log4j-1.2-api', version: '2.6.2'
}
```

Apache Commons Logging Bridge プロジェクトでlog4j 1.xをするApache Commons Loggingをしていて、それをlog4j 2.xにするは、のをします。

Mavenビルド

```
<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-jcl</artifactId>
    <version>2.6.2</version>
  </dependency>
</dependencies>
```

アイビービルド

```
<dependencies>
  <dependency org="org.apache.logging.log4j" name="log4j-jcl" rev="2.6.2" />
</dependencies>
```

Gradle Build

```
dependencies {
  compile group: 'org.apache.logging.log4j', name: 'log4j-jcl', version: '2.6.2'
}
```

Apacheコモンログののをしないでください

リファレンス <https://logging.apache.org/log4j/2.x/maven-artifacts.html>

プロパティ - DBにログするファイル

このをするには、データベースがしているシステムとのあるJDBCドライバがです。 IBM System iのDB2データベースにできるオープンソースのものは、ここにあります [JT400](#)

これはDB2のものですが、ドライバをしてJDBC URLをさせると、ほとんどのシステムでします。

```
# Root logger option
log4j.rootLogger= ERROR, DB
```

```
# Redirect log messages to a DB2
# Define the DB appender
log4j.appender.DB=org.apache.log4j.jdbc.JDBCAppender

# Set JDBC URL (!!! adapt to your target system !!!)
log4j.appender.DB.URL=jdbc:as400://10.10.10.1:446/DATABASENAME;naming=system;errors=full;

# Set Database Driver (!!! adapt to your target system !!!)
log4j.appender.DB.driver=com.ibm.as400.access.AS400JDBCdriver

# Set database user name and password
log4j.appender.DB.user=USER
log4j.appender.DB.password=PASSWORD

# Set the SQL statement to be executed.
log4j.appender.DB.sql=INSERT INTO DB.TABLENAME VALUES ('%d{yyyy-MM-dd}', '%d{HH:mm:ss}', '%C', '%p', '%m')

# Define the layout for file appender
log4j.appender.DB.layout=org.apache.log4j.PatternLayout
```

ログをレベルにフィルタリングするlog4j 1.x

フィルターをして、たとえば`ERROR`レベルよりも「い」メッセージのみをすることができます。しかし、`Filter`は`PropertyConfigurator`ではサポートされていません。したがって、**XML**をするには、**XML**にするがあります。 [フィルタ](#)については[log4j-Wiki](#)をしてください。

"のレベル"

```
<appender name="info-out" class="org.apache.log4j.FileAppender">
  <param name="File" value="info.log"/>
  <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="%m%n"/>
  </layout>
  <filter class="org.apache.log4j.varia.LevelMatchFilter">
    <param name="LevelToMatch" value="info" />
    <param name="AcceptOnMatch" value="true"/>
  </filter>
  <filter class="org.apache.log4j.varia.DenyAllFilter" />
</appender>
```

または「レベル」

```
<appender name="info-out" class="org.apache.log4j.FileAppender">
  <param name="File" value="info.log"/>
  <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="%m%n"/>
  </layout>
  <filter class="org.apache.log4j.varia.LevelRangeFilter">
    <param name="LevelMax" value="info"/>
    <param name="LevelMin" value="info"/>
    <param name="AcceptOnMatch" value="true"/>
  </filter>
</appender>
```

オンラインでlog4j / log4j2をむ <https://riptutorial.com/ja/java/topic/2472/log4j---log4j2>

54: MultiThreadedアプリケーションでThreadPoolExecutorをする

き

パフォーマンスとデータのアプリケーションをする、のかかるタスクをでし、のタスクをにする
とににちます。このトピックでは、ThreadPoolExecutorsをしてのタスクをにするをします。

Examples

クラス・インスタンスをしてりかなタスクの

のアプリケーションでは、にトリガされ、りてられたタスクのにされるのタイプをすのない、い
わゆる "FireForget"タスクをすることがあります。いファイルのパーシ、ログのローテーション、
。

このでは、Runnableインターフェイスをするクラスと、mainメソッドをむクラスの2つのクラス
をします。

AsyncMaintenanceTaskCompleter.java

```
import lombok.extern.java.Log;

import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.TimeUnit;

@Log
public class AsyncMaintenanceTaskCompleter implements Runnable {
    private int taskNumber;

    public AsyncMaintenanceTaskCompleter(int taskNumber) {
        this.taskNumber = taskNumber;
    }

    public void run() {
        int timeout = ThreadLocalRandom.current().nextInt(1, 20);
        try {
            log.info(String.format("Task %d is sleeping for %d seconds", taskNumber,
            timeout));
            TimeUnit.SECONDS.sleep(timeout);
            log.info(String.format("Task %d is done sleeping", taskNumber));
        } catch (InterruptedException e) {
            log.warning(e.getMessage());
        }
    }
}
```

AsyncExample1

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class AsyncExample1 {
    public static void main(String[] args){
        ExecutorService executorService = Executors.newCachedThreadPool();
        for(int i = 0; i < 10; i++){
            executorService.execute(new AsyncMaintenanceTaskCompleter(i));
        }
        executorService.shutdown();
    }
}
```

AsyncExample1.mainをすると、のようにされます。

```
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 8 is sleeping for 18 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 6 is sleeping for 4 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 2 is sleeping for 6 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 3 is sleeping for 4 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 9 is sleeping for 14 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 4 is sleeping for 9 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 5 is sleeping for 10 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 0 is sleeping for 7 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 1 is sleeping for 9 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 7 is sleeping for 8 seconds
Dec 28, 2016 2:21:07 PM AsyncMaintenanceTaskCompleter run
INFO: Task 6 is done sleeping
Dec 28, 2016 2:21:07 PM AsyncMaintenanceTaskCompleter run
INFO: Task 3 is done sleeping
Dec 28, 2016 2:21:09 PM AsyncMaintenanceTaskCompleter run
INFO: Task 2 is done sleeping
Dec 28, 2016 2:21:10 PM AsyncMaintenanceTaskCompleter run
INFO: Task 0 is done sleeping
Dec 28, 2016 2:21:11 PM AsyncMaintenanceTaskCompleter run
INFO: Task 7 is done sleeping
Dec 28, 2016 2:21:12 PM AsyncMaintenanceTaskCompleter run
INFO: Task 4 is done sleeping
Dec 28, 2016 2:21:12 PM AsyncMaintenanceTaskCompleter run
INFO: Task 1 is done sleeping
Dec 28, 2016 2:21:13 PM AsyncMaintenanceTaskCompleter run
INFO: Task 5 is done sleeping
Dec 28, 2016 2:21:17 PM AsyncMaintenanceTaskCompleter run
INFO: Task 9 is done sleeping
Dec 28, 2016 2:21:21 PM AsyncMaintenanceTaskCompleter run
INFO: Task 8 is done sleeping

Process finished with exit code 0
```

のにはいくつかすべきがありますが、

1. タスクはなでされませんでした。
2. タスクはにランダムなだけスリープしていたため、びされたにずしもするとはりませんでした。

びしクラス・インスタンスをしてりがなタスクの

しているタスクをし、したらそのタスクのをするがあることがよくあります。

このでは、2つのクラスをします.1つはCallable <T>インターフェースここでTはすタイプをするものと、mainメソッドをむものです。

AsyncValueTypeTaskCompleter.java

```
import lombok.extern.java.Log;

import java.util.concurrent.Callable;
import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.TimeUnit;

@Log
public class AsyncValueTypeTaskCompleter implements Callable<Integer> {
    private int taskNumber;

    public AsyncValueTypeTaskCompleter(int taskNumber) {
        this.taskNumber = taskNumber;
    }

    @Override
    public Integer call() throws Exception {
        int timeout = ThreadLocalRandom.current().nextInt(1, 20);
        try {
            log.info(String.format("Task %d is sleeping", taskNumber));
            TimeUnit.SECONDS.sleep(timeout);
            log.info(String.format("Task %d is done sleeping", taskNumber));
        } catch (InterruptedException e) {
            log.warning(e.getMessage());
        }
        return timeout;
    }
}
```

AsyncExample2.java

```
import lombok.extern.java.Log;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

@Log
public class AsyncExample2 {
    public static void main(String[] args) {
```

```

    ExecutorService executorService = Executors.newCachedThreadPool();
    List<Future<Integer>> futures = new ArrayList<>();
    for (int i = 0; i < 10; i++){
        Future<Integer> submittedFuture = executorService.submit(new
AsyncValueTypeTaskCompleter(i));
        futures.add(submittedFuture);
    }
    executorService.shutdown();
    while(!futures.isEmpty()){
        for(int j = 0; j < futures.size(); j++){
            Future<Integer> f = futures.get(j);
            if(f.isDone()){
                try {
                    int timeout = f.get();
                    log.info(String.format("A task just completed after sleeping for %d
seconds", timeout));
                    futures.remove(f);
                } catch (InterruptedException | ExecutionException e) {
                    log.warning(e.getMessage());
                }
            }
        }
    }
}
}
}

```

AsyncExample2.mainをすると、のようにされます。

```

Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 7 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 8 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 2 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 1 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 4 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 9 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 0 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 6 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 5 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 3 is sleeping
Dec 28, 2016 3:07:16 PM AsyncValueTypeTaskCompleter call
INFO: Task 8 is done sleeping
Dec 28, 2016 3:07:16 PM AsyncExample2 main
INFO: A task just completed after sleeping for 1 seconds
Dec 28, 2016 3:07:17 PM AsyncValueTypeTaskCompleter call
INFO: Task 2 is done sleeping
Dec 28, 2016 3:07:17 PM AsyncExample2 main
INFO: A task just completed after sleeping for 2 seconds
Dec 28, 2016 3:07:17 PM AsyncValueTypeTaskCompleter call
INFO: Task 9 is done sleeping
Dec 28, 2016 3:07:17 PM AsyncExample2 main
INFO: A task just completed after sleeping for 2 seconds

```

```
Dec 28, 2016 3:07:19 PM AsyncValueTypeTaskCompleter call
INFO: Task 3 is done sleeping
Dec 28, 2016 3:07:19 PM AsyncExample2 main
INFO: A task just completed after sleeping for 4 seconds
Dec 28, 2016 3:07:20 PM AsyncValueTypeTaskCompleter call
INFO: Task 0 is done sleeping
Dec 28, 2016 3:07:20 PM AsyncExample2 main
INFO: A task just completed after sleeping for 5 seconds
Dec 28, 2016 3:07:21 PM AsyncValueTypeTaskCompleter call
INFO: Task 5 is done sleeping
Dec 28, 2016 3:07:21 PM AsyncExample2 main
INFO: A task just completed after sleeping for 6 seconds
Dec 28, 2016 3:07:25 PM AsyncValueTypeTaskCompleter call
INFO: Task 1 is done sleeping
Dec 28, 2016 3:07:25 PM AsyncExample2 main
INFO: A task just completed after sleeping for 10 seconds
Dec 28, 2016 3:07:27 PM AsyncValueTypeTaskCompleter call
INFO: Task 6 is done sleeping
Dec 28, 2016 3:07:27 PM AsyncExample2 main
INFO: A task just completed after sleeping for 12 seconds
Dec 28, 2016 3:07:29 PM AsyncValueTypeTaskCompleter call
INFO: Task 7 is done sleeping
Dec 28, 2016 3:07:29 PM AsyncExample2 main
INFO: A task just completed after sleeping for 14 seconds
Dec 28, 2016 3:07:31 PM AsyncValueTypeTaskCompleter call
INFO: Task 4 is done sleeping
Dec 28, 2016 3:07:31 PM AsyncExample2 main
INFO: A task just completed after sleeping for 16 seconds
```

の

のにはいくつかすべきことがあります、

1. `ExecutorService.submit`をびすたびに、でするためにリストにされた`Future`のインスタンスがされました
2. `Future`には`isDone`というメソッドがあり、りをチェックしようとするにタスクがしているかどうかをチェックできます。の`Future.get`メソッドをびすと、タスクがするまでのスレッドがブロックされ、タスクをにすることによってられるくのがになるがあります。
3. `Future`オブジェクトのりをチェックするに、`executorService.shutdown`メソッドがびされました。これはではありませんが、これがであることをすためにこのでわれました。
`executorService.shutdown`メソッドは、に`ExecutorService`にされたタスクのをぎませんが、しいタスクがキューにされるのをぎます。

Lambdaをしたタスクのインライン

れたソフトウェアではコードのがされることがありますが、ラムダをしてコードでタスクをインラインでして、コードのをすとながあります。

このでは、`main`メソッドをむのクラスをします。このメソッドでは、ラムダをして`Callable`と`Runnable <T>`のインスタンスをしてします。

AsyncExample3.java


```

import lombok.extern.java.Log;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.*;

@Log
public class AsyncExample3 {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newCachedThreadPool();
        List<Future<Integer>> futures = new ArrayList<>();
        for(int i = 0; i < 5; i++){
            final int index = i;
            executorService.execute(() -> {
                int timeout = getTimeout();
                log.info(String.format("Runnable %d has been submitted and will sleep for %d
seconds", index, timeout));
                try {
                    TimeUnit.SECONDS.sleep(timeout);
                } catch (InterruptedException e) {
                    log.warning(e.getMessage());
                }
                log.info(String.format("Runnable %d has finished sleeping", index));
            });
            Future<Integer> submittedFuture = executorService.submit(() -> {
                int timeout = getTimeout();
                log.info(String.format("Callable %d will begin sleeping", index));
                try {
                    TimeUnit.SECONDS.sleep(timeout);
                } catch (InterruptedException e) {
                    log.warning(e.getMessage());
                }
                log.info(String.format("Callable %d is done sleeping", index));
                return timeout;
            });
            futures.add(submittedFuture);
        }
        executorService.shutdown();
        while(!futures.isEmpty()){
            for(int j = 0; j < futures.size(); j++){
                Future<Integer> f = futures.get(j);
                if(f.isDone()){
                    try {
                        int timeout = f.get();
                        log.info(String.format("A task just completed after sleeping for %d
seconds", timeout));
                        futures.remove(f);
                    } catch (InterruptedException | ExecutionException e) {
                        log.warning(e.getMessage());
                    }
                }
            }
        }

        public static int getTimeout(){
            return ThreadLocalRandom.current().nextInt(1, 20);
        }
    }
}

```

のにはいくつかすべきことがあります、

1. ラムダは、がされているスコープでなとメソッドにアクセスできますが、すべてののはラムダですのためにまたはにでなければなりません。
2. ラムダがCallableかRunnable <T>かをにするはなく、りのはりのによってにされます。

オンラインでMultiThreadedアプリケーションでThreadPoolExecutorをするをむ

<https://riptutorial.com/ja/java/topic/8646/multithreadedアプリケーションでthreadpoolexecutorをする>

55: Nashorn JavaScriptエンジン

き

Nashornは、OracleによってJavaでされたJavaScriptエンジンであり、Java 8でリリースされています。**Nashorn**は、JSR-223をしてJavaアプリケーションにJavaScriptをみみ、スタンドアロンのJavaScriptアプリケーションをすることをにし、**よりれた**パフォーマンスとECMAされたJavaScript。

- `ScriptEngineManager` // `ScriptEngine`クラスのとインストールのメカニズムをします。SPI Service Provider Interfaceをして、
- `ScriptEngineManager.ScriptEngineManager`//コンストラクタ
- `ScriptEngine` //スクリプトへのインターフェイスをする
- `ScriptEngine` `ScriptEngineManager.getEngineByName(String shortName)`//されたのファクトリメソッド
- `Object` `ScriptEngine.eval(String script)`//されたスクリプトをします。
- `Object` `ScriptEngine.evalReader(Reader reader)`//されたソースからスクリプトをロードしてします。
- `ScriptContext` `ScriptEngine.getContext()`//デフォルトのバインディング、リーダー、ライターのプロバイダをします。
- `void` `ScriptContext.setWriter(Writer writer)`//スクリプトをするをします。

NashornはJavaでかれ、Java 8にまれるJavaScriptエンジンです。なものはすべて `javax.script` パッケージにバンドルされています。

`ScriptEngineManager`はAPIがされているため、さまざまなスクリプトJavaScriptのみならず Nashornだけでなくのスクリプトエンジンをできます。

Examples

グローバルをする

```
// Obtain an instance of JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

// Define a global variable
engine.put("textToPrint", "Data defined in Java.");

// Print the global variable
try {
    engine.eval("print(textToPrint);");
} catch (ScriptException ex) {
    ex.printStackTrace();
}

// Outcome:
```

```
// 'Data defined in Java.' printed on standard output
```

ハロー.ナシヨーン

```
// Obtain an instance of JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

// Execute an hardcoded script
try {
    engine.eval("print('Hello Nashorn!');");
} catch (ScriptException ex) {
    // This is the generic Exception subclass for the Scripting API
    ex.printStackTrace();
}

// Outcome:
// 'Hello Nashorn!' printed on standard output
```

JavaScript ファイルをする

```
// Required imports
import javax.script.ScriptEngineManager;
import javax.script.ScriptEngine;
import javax.script.ScriptException;
import java.io.FileReader;
import java.io.FileNotFoundException;

// Obtain an instance of the JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

// Load and execute a script from the file 'demo.js'
try {
    engine.eval(new FileReader("demo.js"));
} catch (FileNotFoundException ex) {
    ex.printStackTrace();
} catch (ScriptException ex) {
    // This is the generic Exception subclass for the Scripting API
    ex.printStackTrace();
}

// Outcome:
// 'Script from file!' printed on standard output
```

demo.js

```
print('Script from file!');
```

スクリプトをする

```
// Obtain an instance of JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");
```

```
// Setup a custom writer
StringWriter stringWriter = new StringWriter();
// Modify the engine context so that the custom writer is now the default
// output writer of the engine
engine.getContext().setWriter(stringWriter);

// Execute some script
try {
    engine.eval("print('Redirected text!');");
} catch (ScriptException ex) {
    ex.printStackTrace();
}

// Outcome:
// Nothing printed on standard output, but
// stringWriter.toString() contains 'Redirected text!'
```

をする

```
// Obtain an instance of JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("JavaScript");

//String to be evaluated
String str = "3+2*4+5";
//Value after doing Arithmetic operation with operator precedence will be 16

//Printing the value
try {
    System.out.println(engine.eval(str));
} catch (ScriptException ex) {
    ex.printStackTrace();
}

//Outcome:
//Value of the string after arithmetic evaluation is printed on standard output.
//In this case '16.0' will be printed on standard output.
```

NashornのJavaScriptでのJavaオブジェクトの

NashornエンジンにJavaオブジェクトをして、Javaコードですることはです。に、いくつかのJavaScriptおよびNashornのがあり、Javaオブジェクトでどのようにするかはずしもではありません。

に、JavaScriptのネイティブJavaオブジェクトのをするがあります。

テストみの

1. ifの。 ifのJSでは、Javaにとってboolであるはありません。これは、のnull、undefined、0、などにしてfalseとされます。
2. ステートメントごとにNashornには、それぞれなるJSおよびJavaオブジェクトにしてりすことができるなのループがあります。
3. オブジェクトのサイズをします。 JSオブジェクトでは、またはのサイズをすプロパティの

さがあります。

タイプ	もし	それぞれ	.length
Java null		なし	
Javaの		なし	0
Java		をりしする	のさ
Java/ロング	= 0	なし	
Java ArrayList		をする	リストのさ
Javaハッシュマップ		をする	ヌル
Java HashSet		アイテムをする	

- がnullでなく、でないかどうかをするには、 `if (some_string)` をすることをおめします
- `for each` がすべてのコレクションをりしするためににでき、コレクションがiterable、nullまたはundefinedでないはをしません
- オブジェクトのさをするに、nullまたはがチェックされていなければなりませんメソッドのびしやJavaオブジェクトのプロパティののはじです

スクリプトからのインターフェースの

```
import java.io.FileReader;
import java.io.IOException;

import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class InterfaceImplementationExample {
    public static interface Pet {
        public void eat();
    }

    public static void main(String[] args) throws IOException {
        // Obtain an instance of JavaScript engine
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("nashorn");

        try {
            //evaluate a script
            /* pet.js */
            /*
                var Pet = Java.type("InterfaceImplementationExample.Pet");

                new Pet() {
                    eat: function() { print("eat"); }
                }
            */
        }
    }
}
```

```

        */

        Pet pet = (Pet) engine.eval(new FileReader("pet.js"));

        pet.eat();
    } catch (ScriptException ex) {
        ex.printStackTrace();
    }

    // Outcome:
    // 'eat' printed on standard output
}
}

```

グローバルのと

```

// Obtain an instance of JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

try {
    // Set value in the global name space of the engine
    engine.put("name", "Nashorn");
    // Execute an hardcoded script
    engine.eval("var value='Hello ' + name + '!';");
    // Get value
    String value = (String) engine.get("value");
    System.out.println(value);
} catch (ScriptException ex) {
    // This is the generic Exception subclass for the Scripting API
    ex.printStackTrace();
}

// Outcome:
// 'Hello Nashorn!' printed on standard output

```

オンラインでNashorn JavaScriptエンジンをむ <https://riptutorial.com/ja/java/topic/166/nashorn-javascriptエンジン>

56: NIO - ネットワーク

`SelectionKey`は、`Selector`と`Channel`のでなるなとをします。に、をするために`ファイル`をすることができます。

`OP_READ`はです。ただし、`OP_WRITE`をうときはが`OP_WRITE`。ほとんどの、データはソケットにきまれるため、イベントはしけます。`OP_WRITE`は、データをきむにすることがあります `それを`。

また、`OP_CONNECT`チャンネルがされていたらまあ、それがされている、ので。をしてください。キャンセルするがある`これを`し、`その`ように。したがって、`OP_CONNECT` `finishConnect()`の`OP_CONNECT`はしました。

Examples

セクタをしてイベントをする`OP_CONNECT`の

NIOはJava 1.4にし、のI/Oよりもいとわれる「チャネル」のをしました。ネットワークでは、`SelectableChannel`はチャンネルのさまざまなをできるので、いいものです。これは、`C select()`システムコールとじようにします。ののイベントがしたときにきる

- した `OP_ACCEPT`
- がされた `OP_CONNECT`
- リードFIFO `OP_READ` でなデータ
- データをFIFO `OP_WRITE` にきむことができます。

それは、ソケットI/Oのかがみき...とI/Oみみ/きみ/...をすることののをにします。に、すべてのI/Oはのソケットクライアントにして1つのスレッドでできますが、I/Oのはスレッドプールやそののでできます。これにより、アプリケーションをされたクライアントののにすることができます。

のは、をしています。

1. `Selector`する
2. `SocketChannel`
3. `Selector` `SocketChannel`をする
4. `Selector`をしてイベントをするループ

```
Selector sel = Selector.open(); // Create the Selector
SocketChannel sc = SocketChannel.open(); // Create a SocketChannel
sc.configureBlocking(false); // ... non blocking
sc.setOption(StandardSocketOptions.SO_KEEPALIVE, true); // ... set some options

// Register the Channel to the Selector for wake-up on CONNECT event and use some description
// as an attachment
sc.register(sel, SelectionKey.OP_CONNECT, "Connection to google.com"); // Returns a
SelectionKey: the association between the SocketChannel and the Selector
```



```

System.out.println("Initiating connection");
if (sc.connect(new InetSocketAddress("www.google.com", 80)))
    System.out.println("Connected"); // Connected right-away: nothing else to do
else {
    boolean exit = false;
    while (!exit) {
        if (sel.select(100) == 0) // Did something happen on some registered Channels during
the last 100ms?
            continue; // No, wait some more

        // Something happened...
        Set<SelectionKey> keys = sel.selectedKeys(); // List of SelectionKeys on which some
registered operation was triggered
        for (SelectionKey k : keys) {
            System.out.println("Checking "+k.attachment());
            if (k.isConnectable()) { // CONNECT event
                System.out.print("Connected through select() on "+k.channel()+" -> ");
                if (sc.finishConnect()) { // Finish connection process
                    System.out.println("done!");
                    k.interestOps(k.interestOps() & ~SelectionKey.OP_CONNECT); // We are
already connected: remove interest in CONNECT event
                    exit = true;
                } else
                    System.out.println("unfinished...");
            }
            // TODO: else if (k.isReadable()) { ...
        }
        keys.clear(); // Have to clear the selected keys set once processed!
    }
}
System.out.print("Disconnecting ... ");
sc.shutdownOutput(); // Initiate graceful disconnection
// TODO: empty receive buffer
sc.close();
System.out.println("done");

```

のようにされます。

```

Initiating connection
Checking Connection to google.com
Connected through 'select()' on java.nio.channels.SocketChannel[connection-pending
remote=www.google.com/216.58.208.228:80] -> done!
Disconnecting ... done

```

オンラインでNIO - ネットワークをむ <https://riptutorial.com/ja/java/topic/5513/nio----ネットワーク>

57: NumberFormat

Examples

NumberFormat

なるではかなり、これをすると、JavaのLocaleをしてなるをできます。ロケールをするとにちます

```
Locale locale = new Locale("en", "IN");
NumberFormat numberFormat = NumberFormat.getInstance(locale);
```

のフォーマットをすると、さまざまなタスクをできます

1. フォーマット

```
numberFormat.format(10000000.99);
```

2. フォーマット

```
NumberFormat currencyFormat = NumberFormat.getCurrencyInstance(locale);
currencyFormat.format(10340.999);
```

3. フォーマットパーセンテージ

```
NumberFormat percentageFormat = NumberFormat.getPercentInstance(locale);
percentageFormat.format(10929.999);
```

4.

```
numberFormat.setMinimumIntegerDigits(int digits)
numberFormat.setMaximumIntegerDigits(int digits)
numberFormat.setMinimumFractionDigits(int digits)
numberFormat.setMaximumFractionDigits(int digits)
```

オンラインでNumberFormatをむ <https://riptutorial.com/ja/java/topic/7399/numberformat>

58: RSA

Examples

OAEPとGCMからなるハイブリッドをいた

のでは、AES GCMとOAEPでされる[ハイブリッド](#)をして、デフォルトのパラメータサイズと128ビットのAESキーサイズをしてデータをしています。

OAEPはPKCS1 v1.5のパディングよりもパディング・オラクルのをけにくいです。GCMはパディング・オラクルからもされます。

は、カプセルされたキーのさをにし、にカプセルされたキーをすることによってできます。カプセルされたは、とペアをするRSAをしてすることができます。その、AES / GCMでされたをのにすることができます。

このプロトコルは、

1. ラップされたキーのさフィールド `RSAPrivateKey`は`getKeySize()`メソッドをす。
2. ラップされた/カプセルされたキーRSAキーサイズとしサイズ
3. GCMと128ビットタグJavaによってにされます。

ノート

- このコードをしくするには、なくとも2048ビットのRSAキーをするがありますが、きいがいですがにはくなります。
- AES-256をするには、まず[のポリシーファイル](#)をインストールするがあります。
- のプロトコルをするわりに、わりにCryptographic Message SyntaxCMS / PKCS7やPGPなどのコンテナをすることもできます。

ここにがあります

```
/**
 * Encrypts the data using a hybrid crypto-system which uses GCM to encrypt the data and OAEP
 * to encrypt the AES key.
 * The key size of the AES encryption will be 128 bit.
 * All the default parameter choices are used for OAEP and GCM.
 *
 * @param publicKey the RSA public key used to wrap the AES key
 * @param plaintext the plaintext to be encrypted, not altered
 * @return the ciphertext
 * @throws InvalidKeyException if the key is not an RSA public key
 * @throws NullPointerException if the plaintext is null
 */
public static byte[] encryptData(PublicKey publicKey, byte[] plaintext)
    throws InvalidKeyException, NullPointerException {

    // --- create the RSA OAEP cipher ---
```

```

Cipher oaep;
try {
    // SHA-1 is the default and not vulnerable in this setting
    // use OAEPParameterSpec to configure more than just the hash
    oaep = Cipher.getInstance("RSA/ECB/OAEPwithSHA1andMGF1Padding");
} catch (NoSuchAlgorithmException e) {
    throw new RuntimeException(
        "Runtime doesn't have support for RSA cipher (mandatory algorithm for
runtimes)", e);
} catch (NoSuchPaddingException e) {
    throw new RuntimeException(
        "Runtime doesn't have support for OAEP padding (present in the standard Java
runtime sinze XX)", e);
}
oaep.init(Cipher.WRAP_MODE, publicKey);

// --- wrap the plaintext in a buffer

// will throw NullPointerException if plaintext is null
ByteBuffer plaintextBuffer = ByteBuffer.wrap(plaintext);

// --- generate a new AES secret key ---

KeyGenerator aesKeyGenerator;
try {
    aesKeyGenerator = KeyGenerator.getInstance("AES");
} catch (NoSuchAlgorithmException e) {
    throw new RuntimeException(
        "Runtime doesn't have support for AES key generator (mandatory algorithm for
runtimes)", e);
}
// for AES-192 and 256 make sure you've got the rights (install the
// Unlimited Crypto Policy files)
aesKeyGenerator.init(128);
SecretKey aesKey = aesKeyGenerator.generateKey();

// --- wrap the new AES secret key ---

byte[] wrappedKey;
try {
    wrappedKey = oaep.wrap(aesKey);
} catch (IllegalBlockSizeException e) {
    throw new RuntimeException(
        "AES key should always fit OAEP with normal sized RSA key", e);
}

// --- setup the AES GCM cipher mode ---

Cipher aesGCM;
try {
    aesGCM = Cipher.getInstance("AES/GCM/NoPadding");
    // we can get away with a zero nonce since the key is randomly generated
    // 128 bits is the recommended (maximum) value for the tag size
    // 12 bytes (96 bits) is the default nonce size for GCM mode encryption
    GCMParameterSpec staticParameterSpec = new GCMParameterSpec(128, new byte[12]);
    aesGCM.init(Cipher.ENCRYPT_MODE, aesKey, staticParameterSpec);
} catch (NoSuchAlgorithmException e) {
    throw new RuntimeException(
        "Runtime doesn't have support for AES cipher (mandatory algorithm for
runtimes)", e);
}

```

```

    } catch (NoSuchPaddingException e) {
        throw new RuntimeException(
            "Runtime doesn't have support for GCM (present in the standard Java runtime
sinze XX)", e);
    } catch (InvalidAlgorithmParameterException e) {
        throw new RuntimeException(
            "IvParameterSpec not accepted by this implementation of GCM", e);
    }

    // --- create a buffer of the right size for our own protocol ---

    ByteBuffer ciphertextBuffer = ByteBuffer.allocate(
        Short.BYTES
        + oaep.getOutputSize(128 / Byte.SIZE)
        + aesGCM.getOutputSize(plaintext.length));

    // - element 1: make sure that we know the size of the wrapped key
    ciphertextBuffer.putShort((short) wrappedKey.length);

    // - element 2: put in the wrapped key
    ciphertextBuffer.put(wrappedKey);

    // - element 3: GCM encrypt into buffer
    try {
        aesGCM.doFinal(plaintextBuffer, ciphertextBuffer);
    } catch (ShortBufferException | IllegalBlockSizeException | BadPaddingException e) {
        throw new RuntimeException("Cryptographic exception, AES/GCM encryption should not
fail here", e);
    }

    return ciphertextBuffer.array();
}

```

もちろん、はなしではそれほどではありません。がした、これはのをすことにしてください。

```

/**
 * Decrypts the data using a hybrid crypto-system which uses GCM to encrypt
 * the data and OAEP to encrypt the AES key. All the default parameter
 * choices are used for OAEP and GCM.
 *
 * @param privateKey
 *         the RSA private key used to unwrap the AES key
 * @param ciphertext
 *         the ciphertext to be encrypted, not altered
 * @return the plaintext
 * @throws InvalidKeyException
 *         if the key is not an RSA private key
 * @throws NullPointerException
 *         if the ciphertext is null
 * @throws IllegalArgumentException
 *         with the message "Invalid ciphertext" if the ciphertext is invalid (minimize
information leakage)
 */
public static byte[] decryptData(PublicKey publicKey, byte[] ciphertext)
    throws InvalidKeyException, NullPointerException {

    // --- create the RSA OAEP cipher ---

    Cipher oaep;
    try {

```

```

        // SHA-1 is the default and not vulnerable in this setting
        // use OAEPParameterSpec to configure more than just the hash
        oaep = Cipher.getInstance("RSA/ECB/OAEPwithSHA1andMGF1Padding");
    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException(
            "Runtime doesn't have support for RSA cipher (mandatory algorithm for
runtimes)",
            e);
    } catch (NoSuchPaddingException e) {
        throw new RuntimeException(
            "Runtime doesn't have support for OAEP padding (present in the standard Java
runtime sinze XX)",
            e);
    }
    oaep.init(Cipher.UNWRAP_MODE, privateKey);

    // --- wrap the ciphertext in a buffer

    // will throw NullPointerException if ciphertext is null
    ByteBuffer ciphertextBuffer = ByteBuffer.wrap(ciphertext);

    // sanity check #1
    if (ciphertextBuffer.remaining() < 2) {
        throw new IllegalArgumentException("Invalid ciphertext");
    }
    // - element 1: the length of the encapsulated key
    int wrappedKeySize = ciphertextBuffer.getShort() & 0xFFFF;
    // sanity check #2
    if (ciphertextBuffer.remaining() < wrappedKeySize + 128 / Byte.SIZE) {
        throw new IllegalArgumentException("Invalid ciphertext");
    }

    // --- unwrap the AES secret key ---

    byte[] wrappedKey = new byte[wrappedKeySize];
    // - element 2: the encapsulated key
    ciphertextBuffer.get(wrappedKey);
    SecretKey aesKey;
    try {
        aesKey = (SecretKey) oaep.unwrap(wrappedKey, "AES",
            Cipher.SECRET_KEY);
    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException(
            "Runtime doesn't have support for AES cipher (mandatory algorithm for
runtimes)",
            e);
    } catch (InvalidKeyException e) {
        throw new RuntimeException(
            "Invalid ciphertext");
    }

    // --- setup the AES GCM cipher mode ---

    Cipher aesGCM;
    try {
        aesGCM = Cipher.getInstance("AES/GCM/Nopadding");
        // we can get away with a zero nonce since the key is randomly
        // generated
        // 128 bits is the recommended (maximum) value for the tag size
        // 12 bytes (96 bits) is the default nonce size for GCM mode
        // encryption

```

```

        GCMParameterSpec staticParameterSpec = new GCMParameterSpec(128,
            new byte[12]);
        aesGCM.init(Cipher.DECRYPT_MODE, aesKey, staticParameterSpec);
    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException(
            "Runtime doesn't have support for AES cipher (mandatory algorithm for
runtimes)",
            e);
    } catch (NoSuchPaddingException e) {
        throw new RuntimeException(
            "Runtime doesn't have support for GCM (present in the standard Java runtime
sinze XX)",
            e);
    } catch (InvalidAlgorithmParameterException e) {
        throw new RuntimeException(
            "IvParameterSpec not accepted by this implementation of GCM",
            e);
    }

    // --- create a buffer of the right size for our own protocol ---

    ByteBuffer plaintextBuffer = ByteBuffer.allocate(aesGCM
        .getOutputSize(ciphertextBuffer.remaining()));

    // - element 3: GCM ciphertext
    try {
        aesGCM.doFinal(ciphertextBuffer, plaintextBuffer);
    } catch (ShortBufferException | IllegalBlockSizeException
        | BadPaddingException e) {
        throw new RuntimeException(
            "Invalid ciphertext");
    }

    return plaintextBuffer.array();
}

```

オンラインでRSAをむ <https://riptutorial.com/ja/java/topic/1889/rsa>

59: ServiceLoader

`ServiceLoader`をして、`.jar`ファイルにパックされたファイルでされた、された=サービスをするクラスのインスタンスをできます。/されたサービスはくのインタフェースですが、これはではありません。

/クラスは、`ServiceLoader`がそれらをインスタンスするためのゼロコンストラクタをするがあります。

`ServiceLoader`によってされるには、されたサービスののをつテキストファイルを、`jar`ファイルの `META-INF/services` ディレクトリにするがあります。このファイルには、1にサービスをするクラスのが1つまれています。

Examples

ロガーサービス

のは、`ServiceLoader`をしてログをするためにクラスをインスタンスするをしています。

サービス

```
package servicetest;

import java.io.IOException;

public interface Logger extends AutoCloseable {

    void log(String message) throws IOException;

}
```

サービスの

のは、にメッセージを `System.err` きみます

```
package servicetest.logger;

import servicetest.Logger;

public class ConsoleLogger implements Logger {

    @Override
    public void log(String message) {
        System.err.println(message);
    }

    @Override
    public void close() {
```



```
}  
  
}
```

のでは、メッセージをテキストファイルにきみます。

```
package servicetest.logger;  
  
import java.io.BufferedWriter;  
import java.io.FileWriter;  
import java.io.IOException;  
import servicetest.Logger;  
  
public class FileLogger implements Logger {  
  
    private final BufferedWriter writer;  
  
    public FileLogger() throws IOException {  
        writer = new BufferedWriter(new FileWriter("log.txt"));  
    }  
  
    @Override  
    public void log(String message) throws IOException {  
        writer.append(message);  
        writer.newLine();  
    }  
  
    @Override  
    public void close() throws IOException {  
        writer.close();  
    }  
  
}
```

META-INF / services / servicetest.Logger

META-INF/services/servicetest.Logger ファイルには、Logger のがリストされています。

```
servicetest.logger.ConsoleLogger  
servicetest.logger.FileLogger
```

の`main`メソッドは、なすすべてのロガーにメッセージをきみます。ロガーは`ServiceLoader`をしてインスタンスされます。

```
public static void main(String[] args) throws Exception {  
    final String message = "Hello World!";  
  
    // get ServiceLoader for Logger  
    ServiceLoader<Logger> loader = ServiceLoader.load(servicetest.Logger.class);  
  
    // iterate through instances of available loggers, writing the message to each one  
    Iterator<Logger> iterator = loader.iterator();
```

```

while (iterator.hasNext()) {
    try (Logger logger = iterator.next()) {
        logger.log(message);
    }
}
}

```

なServiceLoaderの

ServiceLoaderは、インターフェイスのロードのためのシンプルでいやすいみみのメカニズムです。インスタントのためののをするサービスローダーただしはありませんでは、Java SEになをみむことができます。ServiceLoaderインターフェースとのはになり、プログラムはにできます。、くのJava APIはServiceLoaderについています

なコンセプトは

- サービスのインターフェイスでの
- ServiceLoaderをしてサービスののをする
- サービスののをする

インタフェースからまり、例えばaccounting-api.jarというのjar accounting-api.jarれてみましょう。

```

package example;

public interface AccountingService {

    long getBalance();
}

```

に、accounting-impl.jarというのjar accounting-impl.jarにサービスののをします。これにはサービスのがまれています

```

package example.impl;
import example.AccountingService;

public interface DefaultAccountingService implements AccountingService {

    public long getBalance() {
        return balanceFromDB();
    }

    private long balanceFromDB(){
        ...
    }
}

```

さらに、accounting-impl.jarは、このjarがAccountingServiceをすることをするファイルがまれています。ファイルは、META-INF/services/まるパスをつがあり、とじをつがあります。

- META-INF/services/example.AccountingService

ファイルのは、のです。

```
example.impl.DefaultAccountingService
```

のjarがプログラムのクラスパスにあるとすると、それはAccountingServiceをし、サービスのインスタンスはServiceLauncherをしてできます

```
ServiceLoader<AccountingService> loader = ServiceLoader.load(AccountingService.class)
AccountingService service = loader.next();
long balance = service.getBalance();
```

ServiceLoaderはIterable、のプロバイダがサポートされています。プログラムのはのとおりで
。

```
ServiceLoader<AccountingService> loader = ServiceLoader.load(AccountingService.class)
for(AccountingService service : loader) {
    //...
}
```

next()をびすと、しいインスタンスがにされることにしてください。インスタンスをするは、の
ようにServiceLoaderまたはfor-eachループのiterator()メソッドをするがありiterator()。

オンラインでServiceLoaderをむ <https://riptutorial.com/ja/java/topic/5433/serviceloader>

60: SortedMap

き

ソートマップの。

Examples

ソートマップの。

キーポイント -

- SortedMap インタフェースは Map をします。
- エントリはのキーにされます。

ソートされたマップのメソッド

- コンパレータ コンパレータ。
- オブジェクト firstKey。
- SortedMap headMap オブジェクト。
- オブジェクト lastKey。
- SortedMap subMap オブジェクト、オブジェクト。
- SortedMap tailMapObject start

```
public static void main(String args[]) {
    // Create a hash map
    TreeMap tm = new TreeMap();

    // Put elements to the map
    tm.put("Zara", new Double(3434.34));
    tm.put("Mahnaz", new Double(123.22));
    tm.put("Ayan", new Double(1378.00));
    tm.put("Daisy", new Double(99.22));
    tm.put("Qadir", new Double(-19.08));

    // Get a set of the entries
    Set set = tm.entrySet();

    // Get an iterator
    Iterator i = set.iterator();

    // Display elements
    while(i.hasNext()) {
        Map.Entry me = (Map.Entry)i.next();
        System.out.print(me.getKey() + ": ");
        System.out.println(me.getValue());
    }
    System.out.println();

    // Deposit 1000 into Zara's account
```

```
double balance = ((Double)tm.get("Zara")).doubleValue();  
tm.put("Zara", new Double(balance + 1000));  
System.out.println("Zara's new balance: " + tm.get("Zara"));  
}
```

オンラインでSortedMapをむ <https://riptutorial.com/ja/java/topic/10748/sortedmap>

61: Stack-Walking API

き

Java 9よりでは、スレッドスタックフレームへのアクセスは、クラス `sun.reflect.Reflection` にされていました。には、 `sun.reflect.Reflection::getCallerClass` のライブラリはこのメソッドにしていますが、です。

APIはJDK 9にけられている `java.lang.StackWalker` クラス、およびスタックフレームになアクセスをにすることにより、にされています。のアプリケーションでは、このAPIをしてクラスのスタックとフィルタをトラバースすることがあります。

Examples

のスレッドのすべてのスタックフレームをする

は、のスレッドのすべてのスタックフレームをします。

```
1  package test;
2
3  import java.lang.StackWalker.StackFrame;
4  import java.lang.reflect.InvocationTargetException;
5  import java.lang.reflect.Method;
6  import java.util.List;
7  import java.util.stream.Collectors;
8
9  public class StackWalkerExample {
10
11      public static void main(String[] args) throws NoSuchMethodException, SecurityException,
12      IllegalAccessException, IllegalArgumentException, InvocationTargetException {
13          Method fooMethod = FooHelper.class.getDeclaredMethod("foo", (Class<?>[])null);
14          fooMethod.invoke(null, (Object[]) null);
15      }
16
17      class FooHelper {
18          protected static void foo() {
19              BarHelper.bar();
20          }
21      }
22
23      class BarHelper {
24          protected static void bar() {
25              List<StackFrame> stack = StackWalker.getInstance()
26                  .walk((s) -> s.collect(Collectors.toList()));
27              for(StackFrame frame : stack) {
28                  System.out.println(frame.getClassName() + " " + frame.getLineNumber() + " " +
29                  frame.getMethodName());
30              }
31          }
32      }
33  }
```

```
test.BarHelper 26 bar
test.FooHelper 19 foo
test.StackWalkerExample 13 main
```

のびしクラスをする

は、のびしクラスをします。この、[StackWalker](#) オプション [RETAIN_CLASS_REFERENCE](#) であるため、`Class` インスタンスは [StackFrame](#) オブジェクトにされます。それのはがします。

```
public class StackWalkerExample {

    public static void main(String[] args) {
        FooHelper.foo();
    }

}

class FooHelper {
    protected static void foo() {
        BarHelper.bar();
    }
}

class BarHelper {
    protected static void bar() {

System.out.println(StackWalker.getInstance(Option.RETAIN_CLASS_REFERENCE).getCallerClass());
    }
}
```

```
class test.FooHelper
```

などのしフレームの

いくつかのオプションでは、スタックトレースにフレームやリフレクションフレームをめることができます。これは、デバッグのにちます。たとえば、に [SHOW_REFLECT_FRAMES](#) オプションを [StackWalker](#) インスタンスにすると、メソッドのフレームもにされます。

```
package test;

import java.lang.StackWalker.Option;
import java.lang.StackWalker.StackFrame;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.List;
import java.util.stream.Collectors;

public class StackWalkerExample {

    public static void main(String[] args) throws NoSuchMethodException, SecurityException,
IllegalAccess Exception, IllegalArgumentException, InvocationTargetException {
        Method fooMethod = FooHelper.class.getDeclaredMethod("foo", (Class<?>[])null);
        fooMethod.invoke(null, (Object[]) null);
    }
}
```

```

}

class FooHelper {
    protected static void foo() {
        BarHelper.bar();
    }
}

class BarHelper {
    protected static void bar() {
        // show reflection methods
        List<StackFrame> stack = StackWalker.getInstance(Option.SHOW_REFLECT_FRAMES)
            .walk((s) -> s.collect(Collectors.toList()));
        for(StackFrame frame : stack) {
            System.out.println(frame.getClassName() + " " + frame.getLineNumber() + " " +
frame.getMethodName());
        }
    }
}

```

```

test.BarHelper 27 bar
test.FooHelper 20 foo
jdk.internal.reflect.NativeMethodAccessorImpl -2 invoke0
jdk.internal.reflect.NativeMethodAccessorImpl 62 invoke
jdk.internal.reflect.DelegatingMethodAccessorImpl 43 invoke
java.lang.reflect.Method 563 invoke
test.StackWalkerExample 14 main

```

のリフレクションメソッドのはできないため、`StackFrame.getLineNumber()` はのをすがあることにしてください。

オンラインでStack-Walking APIをむ <https://riptutorial.com/ja/java/topic/9868/stack-walking-api>

62: static キーワードの

- パブリック `static int myVariable; //`をする
- `public static myMethod{} //`メソッドをします。
- パブリック `static final double MY_CONSTANT; //`クラスのすべてのインスタンスでされるをする
- `public final double MY_CONSTANT; //`クラスのこのインスタンスにのをしますインスタンスごとになるをするコンストラクタでもよくされます

Examples

static をしてをする

`static` キーワードは、インスタンスされたクラスのないフィールドやメソッドにアクセスするためにされるため、のクラスでするをするためにできます。これらは、クラスのインスタンスのたびにのまます。、 `static` はに `ALL_CAPS` で、ラクダケースではなくアンダースコアをします。

```
static E STATIC_VARIABLE_NAME
```

はできないので、 `static` は `final` でもできます。

たとえば、`pi` のをするには

```
public class MathUtilities {  
  
    static final double PI = 3.14159265358  
  
}
```

のクラスでとしてできます。たとえば、のようになります。

```
public class MathCalculations {  
  
    //Calculates the circumference of a circle  
    public double calculateCircumference(double radius) {  
        return (2 * radius * MathUtilities.PI);  
    }  
  
}
```

これでをう

`Static` は、クラスのインスタンスにりてられていないメソッドまたはのをします。むしろ、はすべてのクラスメンバーでされます。ちなみに、をクラスインスタンスのメンバーのようにうと、がされます。

```
public class Apple {
    public static int test;
    public int test2;
}

Apple a = new Apple();
a.test = 1; // Warning
Apple.test = 1; // OK
Apple.test2 = 1; // Illegal: test2 is not static
a.test2 = 1; // OK
```

とされたメソッドは、じようにしますが、のがあります。

`this` キーワードをすることはできません

```
public class Pineapple {

    private static int numberOfSpikes;
    private int age;

    public static getNumberOfSpikes() {
        return this.numberOfSpikes; // This doesn't compile
    }

    public static getNumberOfSpikes() {
        return numberOfSpikes; // This compiles
    }

}
```

には、クラスとなるインスタンスクローンメソッドなどにされるジェネリックメソッドを`static`としてし、`equals()`などのメソッド`equals()`メソッドとしてください。Javaプログラムの`main`はにです。つまり、`this` キーワードを`main()` することはできません。

コンテキストからのメンバーへの

とメソッドはインスタンスではありません。のクラスのオブジェクトをいくつしても、そのコピーがに1つあります。

えば、のなリストをつことができます。メソッドのにだけにするとよいでしょう。これにより、のクラスのインスタンスをにするに、パフォーマンスがにします。

さらに、クラスにブロックをつこともできます。これをして、にデフォルトをりてることができます。それらは、クラスがメモリにロードされるときにだけされます。

されたとしてのインスタンスは、のオブジェクトのインスタンスにしており、それらは、そのまぐれにちます。あなたは、オブジェクトののライフサイクルのにそれらとにぶことができます。

そのクラスのメソッドでされるクラスのすべてのフィールドおよびメソッドは、またはローカルでなければなりません。インスタンスまたはメソッドをしようとする、コードはコンパイルされません。

```
public class Week {
    static int daysOfTheWeek = 7; // static variable
    int dayOfTheWeek; // instance variable

    public static int getDaysLeftInWeek(){
        return Week.daysOfTheWeek-dayOfTheWeek; // this will cause errors
    }

    public int getDaysLeftInWeek(){
        return Week.daysOfTheWeek-dayOfTheWeek; // this is valid
    }

    public static int getDaysLeftInTheWeek(int today){
        return Week.daysOfTheWeek-today; // this is valid
    }
}
```

オンラインでstaticキーワードのをむ <https://riptutorial.com/ja/java/topic/2253/static>キーワードの

63: StringBuffer

き

Java **StringBuffer** クラスの。

Examples

String Buffer クラス

キーポイント -

- ななをするためにされます。
- - です。
- スレッドセーフです。つまり、のスレッドがにアクセスすることはできません。

-

- パブリック **StringBuffer appendString s**
- パブリックされた **StringBuffer int offset、String s**
- パブリックされた **StringBuffer replace int startIndex、int endIndex、String str**
- パブリックされた **StringBuffer delete int startIndex、int endIndex**
- **public synchronized StringBuffer reverse**
- **public int capacity**
- **public void ensureCapacity int minimumCapacity**
- パブリック **char charAt int index**
- **public int length**
- **public String substring int beginIndex**
- パブリック **String substring int beginIndex、int endIndex**

とのの **difference** をすバッファ -

```
class Test {  
    public static void main(String args[])  
    {
```

```
String str = "study";  
str.concat("tonight");  
System.out.println(str);          // Output: study  
  
StringBuffer strB = new StringBuffer("study");  
strB.append("tonight");  
System.out.println(strB);        // Output: studytonight  
}  
}
```

オンラインでStringBufferをむ <https://riptutorial.com/ja/java/topic/10757/stringbuffer>

64: StringBuilder

き

Java StringBuilderクラスは、ななをするためにされます。Java StringBuilderクラスは、していな
いをいて、StringBufferクラスとじです。JDK 1.5でできます。

- しいStringBuilder
- しいStringBuilderint
- しいStringBuilderCharSequence seq
- しいStringBuilderStringBuilder Builder
- しいStringBuilder
- しいStringJoinerCharSequenceり
- しいStringJoinerCharSequenceり、CharSequence、CharSequence

charをパラメータとしてしいStringBuilderをすると、int capacityをつコンストラクタがびされ、
String stringはされません。

```
StringBuilder v = new StringBuilder('I'); //'I' is a character, "I" is a String.  
System.out.println(v.capacity()); --> output 73  
System.out.println(v.toString()); --> output nothing
```

Examples

nStringをりします。

StringむnのりしString s。

なアプローチは、りしStringをすることです

```
final int n = ...  
final String s = ...  
String result = "";  
  
for (int i = 0; i < n; i++) {  
    result += s;  
}
```

これは、 n に1をむしいインスタンス n のりし s のにられた $O(s.length() * n^2) = O(s.length() * (1+2+\dots+(n-1)+n))$ 。

これをけるには、 `StringBuilder` をするがあります。これにより、わりに `O(s.length() * n)` `String` をできます。

```
final int n = ...
final String s = ...

StringBuilder builder = new StringBuilder();

for (int i = 0; i < n; i++) {
    builder.append(s);
}

String result = builder.toString();
```

`StringBuffer`、`StringBuilder`、`Formatter`および`StringJoiner`の

`StringBuffer`、`StringBuilder`、`Formatter`および`StringJoiner`クラスは、にのからをアセンブルするためにされるJava SEユーティリティクラスです。

- `StringBuffer`クラスはJava 1.0にし、のをむ「バッファ」をおよびするためのさまざまなメソッドをします。
- の`StringBuffer`クラスのパフォーマンスのにするために、`StringBuilder`クラスがJava 5にされました。2つのクラスターのAPIはにじです。`StringBuffer`と`StringBuilder`のないは、はスレッドセーフであり、はしていないことです。

のは、`StringBuilder`をしています。

```
int one = 1;
String color = "red";
StringBuilder sb = new StringBuilder();
sb.append("One=").append(one).append(", Color=").append(color).append('\n');
System.out.print(sb);
// Prints "One=1, Colour=red" followed by an ASCII newline.
```

`StringBuffer`クラスはじでされますの`StringBuilder`を`StringBuffer`にするだけです

`StringBuffer`と`StringBuilder`クラスは、みておよびのにしています。すなわち、をきえたりしたり、さまざまでそれらをするをします。っている2つのクラスは、をみてるタスクにです。

- `Formatter`クラスはJava 5でされ、Cライブラリの`sprintf`でやかにモデルされています。めみとそのののシーケンスをつをと、をテキストにしてのわりにしてをします。のは、をテキストにするをします。
- `StringJoiner`クラスはJava 8でされました。なのフォーマッタで、のをりで`StringJoiner`フォーマットします。これはなAPIでされており、Java 8ストリームでできます。

`Formatter`なをにします。

```
// This does the same thing as the StringBuilder example above
```

```
int one = 1;
String color = "red";
Formatter f = new Formatter();
System.out.print(f.format("One=%d, colour=%s%n", one, color));
// Prints "One=1, Colour=red" followed by the platform's line separator

// The same thing using the `String.format` convenience method
System.out.print(String.format("One=%d, color=%s%n", one, color));
```

`StringJoiner` クラスはのタスクにはしていないので、ここではのののをします。

```
StringJoiner sj = new StringJoiner(", ", "[", "]");
for (String s : new String[]{"A", "B", "C"}) {
    sj.add(s);
}
System.out.println(sj);
// Prints "[A, B, C]"
```

4つのクラスのユースケースをできます。

- のアセンブリまたはタスクにした `StringBuilder`。
- `StringBuffer` は、スレッドセーフバージョンの `StringBuilder` がなにのみします。
- `Formatter` は、よりなをしますが、 `StringBuilder` ほどではありません。これは、
`Formatter.format(...)` びすたびにのことが `Formatter.format(...)` ため `Formatter.format(...)`。
 - `format` をし、
 - `varargs` のと
 - のプリミティブをオートボクシングする。
- `StringJoiner` は、のシーケンスをセパレータでかつにフォーマットしますが、のフォーマットにはしていません。

オンラインで `StringBuilder` をむ <https://riptutorial.com/ja/java/topic/1037/stringbuilder>

65: sun.misc.Unsafe

`Unsafe` クラスは、プログラムがJavaコンパイラでされていないことをします。のプログラムでは、`Unsafe` をわなないでください。

1. `Unsafe` APIをしてえた、アプリケーションはJVMをクラッシュさせたり、しにくいをしたりするがあります。
2. `Unsafe` APIはなしにされることがあります。コードでするは、Javaバージョンをするときにコードをきすがあります。

Examples

による `sun.misc.Unsafe` のインスタンス

```
public static Unsafe getUnsafe() {
    try {
        Field unsafe = Unsafe.class.getDeclaredField("theUnsafe");
        unsafe.setAccessible(true);
        return (Unsafe) unsafe.get(null);
    } catch (IllegalAccessException e) {
        // Handle
    } catch (IllegalArgumentException e) {
        // Handle
    } catch (NoSuchFieldException e) {
        // Handle
    } catch (SecurityException e) {
        // Handle
    }
}
```

`sun.misc.Unsafe` にはプライベートコンストラクタがあり、な `getUnsafe()` メソッドは、コードがプライマリクラスローダでロードされていることをするためにクラスローダのチェックでされています。したがって、インスタンスをみむ1つののは、をしてフィールドをすることです。

`bootclasspath` で `sun.misc.Unsafe` をインスタンスする

```
public class UnsafeLoader {
    public static Unsafe loadUnsafe() {
        return Unsafe.getUnsafe();
    }
}
```

これはコンパイルされますが、`Unsafe` クラスにプライマリクラスローダがロードされていないり、にするがあります。これをにするために、JVMにはのようなのをロードするがあります。

```
java -Xbootclasspath:$JAVA_HOME/jre/lib/rt.jar:./UnsafeLoader.jar foo.bar.MyApp
```

foo.bar.MyAppクラスはUnsafeLoader.loadUnsafe() できます。

のインスタンスをする

でないのは、アクセスできないプライベートフィールドとしてされます。コンストラクタはprivateであり、public static Unsafe getUnsafe() にアクセスするのメソッドはアクセスです。リフレクションをすると、プライベートフィールドをアクセスにするためのがあります。

```
public static final Unsafe UNSAFE;

static {
    Unsafe unsafe = null;

    try {
        final PrivilegedExceptionAction<Unsafe> action = () -> {
            final Field f = Unsafe.class.getDeclaredField("theUnsafe");
            f.setAccessible(true);

            return (Unsafe) f.get(null);
        };

        unsafe = AccessController.doPrivileged(action);
    } catch (final Throwable t) {
        throw new RuntimeException("Exception accessing Unsafe", t);
    }

    UNSAFE = unsafe;
}
```

の

でないものののはのとおりです。

つかいます	API
ヒープ/ダイレクトメモリりての オフ、りておよびりて	allocateMemory(bytes) 、 reallocateMemory(address, bytes) およびfreeMemory(address)
メモリフェンス	loadFence() 、 storeFence() 、 fullFence()
のスレッドをする	park(isAbsolute, time) 、 unpark(thread)
フィールドおよび/またはメモリ アクセス	メソッドをget*してput*する
チェックされていないのスロー	throwException(e)
CASおよびアトミックオペレー ション	compareAndSwap*メソッドのファミリー
メモリの	setMemory

つかいます	API
または	get*Volatile、 put*Volatile、 putOrdered*

メソッドのgetおよびputファミリーは、されたオブジェクトにしています。オブジェクトがnullの、アドレスとしてわれます。

```
// Putting a value to a field
protected static long fieldOffset = UNSAFE.objectFieldOffset(getClass().getField("theField"));
UNSAFE.putLong(this, fieldOffset, newValue);

// Putting an absolute value
UNSAFE.putLong(null, address, newValue);
UNSAFE.putLong(address, newValue);
```

いくつかのメソッドは、intとlongsにしてのみされます。あなたはとダブルスでこれらのメソッドをすることができます floatToRawIntBits、 intBitsToFloat, doubleToRawLongBits , longBitsToDouble`を

オンラインでsun.misc.Unsafeをむ <https://riptutorial.com/ja/java/topic/6771/sun-misc-unsafe>

66: ThreadLocal

びしをびすににするが、 `SimpleDateFormat`、 `Marshaller` ようにステートレスであるオブジェクトに
のために `Random` に `ThreadLocal` の、 することをしてください `ThreadLocalRandom`

Examples

ThreadLocal Java 8 の

```
public static class ThreadLocalExample
{
    private static final ThreadLocal<SimpleDateFormat> format =
        ThreadLocal.withInitial(() -> new SimpleDateFormat("yyyyMMdd_HHmm"));

    public String formatDate(Date date)
    {
        return format.get().format(date);
    }
}
```

な ThreadLocal の

Java の `ThreadLocal` は、スレッドのローカルをするためにされます。 `Object` のスレッドはをしているため、はスレッドセーフではありません。スレッドのをするためにをすることができますが、をけたいは、 `ThreadLocal` をしてスレッドのローカルなをできます。つまり、そのスレッドのみがそのをみきできるため、それぞれの `ThreadLocal` にアクセスすることはできません。

これは、 `ThreadLocal` をうためにうことができます。たとえば、Web サービスのようなスレッドプールがあるでは、たとえば、リクエストごとに `SimpleDateFormat` オブジェクトをするのはがっかり、 `SimpleDateFormat` はスレッドセーフではないため、スタティクなものをしてできないため、 `SimpleDateFormat` をするオーバーヘッドなしにスレッドセーフなをできるように `ThreadLocal` をできます。

のコードは、どのようにできるかをしています。

すべてのスレッドにはの `ThreadLocal` があり、 `get()` `set()` メソッドと `set()` メソッドをしてデフォルトをしたり、 `Thread` のローカルをすることができます。

`ThreadLocal` インスタンスは、、をスレッドにけるクラスのプライベートフィールドです。

ここでは、Java プログラムで `ThreadLocal` をし、すべてのスレッドに `ThreadLocal` ののコピーがあることをすさなをします。

```
package com.examples.threads;

import java.text.SimpleDateFormat;
```

```

import java.util.Random;

public class ThreadLocalExample implements Runnable{

    // SimpleDateFormat is not thread-safe, so give one to each thread
    // SimpleDateFormat is not thread-safe, so give one to each thread
    private static final ThreadLocal<SimpleDateFormat> formatter = new
ThreadLocal<SimpleDateFormat>(){
    @Override
    protected SimpleDateFormat initialValue()
    {
        return new SimpleDateFormat("yyyyMMdd HHmm");
    }
};

    public static void main(String[] args) throws InterruptedException {
        ThreadLocalExample obj = new ThreadLocalExample();
        for(int i=0 ; i<10; i++){
            Thread t = new Thread(obj, ""+i);
            Thread.sleep(new Random().nextInt(1000));
            t.start();
        }
    }

    @Override
    public void run() {
        System.out.println("Thread Name= "+Thread.currentThread().getName()+" default
Formatter = "+formatter.get().toPattern());
        try {
            Thread.sleep(new Random().nextInt(1000));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        formatter.set(new SimpleDateFormat());

        System.out.println("Thread Name= "+Thread.currentThread().getName()+" formatter =
"+formatter.get().toPattern());
    }
}

```

Thread Name= 0 default Formatter = yyyyMMdd HHmm

Thread Name= 1 default Formatter = yyyyMMdd HHmm

Thread Name= 0 formatter = M/d/yy h:mm a

Thread Name= 2 default Formatter = yyyyMMdd HHmm

Thread Name= 1 formatter = M/d/yy h:mm a

Thread Name= 3 default Formatter = yyyyMMdd HHmm

Thread Name= 4 default Formatter = yyyyMMdd HHmm

Thread Name= 4 formatter = M/d/yy h:mm a

Thread Name= 5 default Formatter = yyyyMMdd HHmm

Thread Name= 2 formatter = M/d/yy h:mm a

```
Thread Name= 3 formatter = M/d/yy h:mm a

Thread Name= 6 default Formatter = yyyyMMdd HHmm

Thread Name= 5 formatter = M/d/yy h:mm a

Thread Name= 6 formatter = M/d/yy h:mm a

Thread Name= 7 default Formatter = yyyyMMdd HHmm

Thread Name= 8 default Formatter = yyyyMMdd HHmm

Thread Name= 8 formatter = M/d/yy h:mm a

Thread Name= 7 formatter = M/d/yy h:mm a

Thread Name= 9 default Formatter = yyyyMMdd HHmm

Thread Name= 9 formatter = M/d/yy h:mm a
```

から、Thread-0はフォーマッタのをしましたが、まだスレッド2のデフォルトフォーマッタはされたとしてであることがわかります。

1つのオブジェクトをつのスレッド

このでは、オブジェクトは1つしかありませんが、なるスレッドで/されています。をするためのフィールドののではありません。なぜなら、のスレッドもそれをたりおそらくえないかもしれないからです。

```
public class Test {
    public static void main(String[] args) {
        Foo foo = new Foo();
        new Thread(foo, "Thread 1").start();
        new Thread(foo, "Thread 2").start();
    }
}
```

Fooではゼロからまります。ステートをフィールドにするわりに、にアクセスなThreadLocalオブジェクトにのをします。こののは、ThreadLocalのとはなく、よりいコンソールをすることにしてください。

```
public class Foo implements Runnable {
    private static final int ITERATIONS = 10;
    private static final ThreadLocal<Integer> threadLocal = new ThreadLocal<Integer>() {
        @Override
        protected Integer initialValue() {
            return 0;
        }
    };

    @Override
    public void run() {
        for (int i = 0; i < ITERATIONS; i++) {
            synchronized (threadLocal) {
                //Although accessing a static field, we get our own (previously saved) value.
                int value = threadLocal.get();
            }
        }
    }
}
```

```

        System.out.println(Thread.currentThread().getName() + ": " + value);

        //Update our own variable
        threadLocal.set(value + 1);

        try {
            threadLocal.notifyAll();
            if (i < ITERATIONS - 1) {
                threadLocal.wait();
            }
        } catch (InterruptedException ex) {
        }
    }
}
}
}

```

から、スレッドはそれぞれのためにカウントされ、のスレッドのはされないことがわかります。

```

Thread 1: 0
Thread 2: 0
Thread 1: 1
Thread 2: 1
Thread 1: 2
Thread 2: 2
Thread 1: 3
Thread 2: 3
Thread 1: 4
Thread 2: 4
Thread 1: 5
Thread 2: 5
Thread 1: 6
Thread 2: 6
Thread 1: 7
Thread 2: 7
Thread 1: 8
Thread 2: 8
Thread 1: 9
Thread 2: 9

```

オンラインでThreadLocalをむ <https://riptutorial.com/ja/java/topic/2001/threadlocal>

67: TreeMapとTreeSet

き

`TreeMap`と`TreeSet`は、Java 1.2でされたなJavaコレクションです。`TreeMap`は、なけされた`Map`です。に、`TreeSet`はなけされた`Set`です。

`TreeMap`はRed-Blackツリーとしてされ、 $O(\log n)$ アクセスをします。`TreeSet`は、ダミーをつ`TreeMap`をしてされています。

どちらのコレクションもスレッドセーフではありません。

Examples

なJavaのTreeMap

まず、のマップをし、いくつかのをします。

Java SE 7

```
TreeMap<Integer, String> treeMap = new TreeMap<>();
```

Java SE 7

```
TreeMap<Integer, String> treeMap = new TreeMap<Integer, String>();
```

```
treeMap.put(10, "ten");
treeMap.put(4, "four");
treeMap.put(1, "one");
treeSet.put(12, "twelve");
```

マップにいくつかのがあれば、いくつかのをできます

```
System.out.println(treeMap.firstEntry()); // Prints 1=one
System.out.println(treeMap.lastEntry()); // Prints 12=twelve
System.out.println(treeMap.size()); // Prints 4, since there are 4 elemens in the map
System.out.println(treeMap.get(12)); // Prints twelve
System.out.println(treeMap.get(15)); // Prints null, since the key is not found in the map
```

また、`Iterator`または`foreach`ループのいずれかをしてマップをすることもできます。エントリは、ではなく、[け](#)によってされることにしてください。

Java SE 7

```
for (Entry<Integer, String> entry : treeMap.entrySet()) {
    System.out.print(entry + " "); //prints 1=one 4=four 10=ten 12=twelve
}
```



```

Iterator<Entry<Integer, String>> iter = treeMap.entrySet().iterator();
while (iter.hasNext()) {
    System.out.print(iter.next() + " "); //prints 1=one 4=four 10=ten 12=twelve
}

```

なJavaのTreeSet

に、のセットをし、いくつかのをします。

Java SE 7

```

TreeSet<Integer> treeSet = new TreeSet<>();

```

Java SE 7

```

TreeSet<Integer> treeSet = new TreeSet<Integer>();

```

```

treeSet.add(10);
treeSet.add(4);
treeSet.add(1);
treeSet.add(12);

```

セットにいくつかのがあれば、いくつかのをできます

```

System.out.println(treeSet.first()); // Prints 1
System.out.println(treeSet.last()); // Prints 12
System.out.println(treeSet.size()); // Prints 4, since there are 4 elemens in the set
System.out.println(treeSet.contains(12)); // Prints true
System.out.println(treeSet.contains(15)); // Prints false

```

また、Iteratorまたはforeachループのいずれかをしてマップをすることもできます。エントリは、ではなく、[け](#)によってされることにしてください。

Java SE 7

```

for (Integer i : treeSet) {
    System.out.print(i + " "); //prints 1 4 10 12
}

```

```

Iterator<Integer> iter = treeSet.iterator();
while (iter.hasNext()) {
    System.out.print(iter.next() + " "); //prints 1 4 10 12
}

```

カスタムJavaのTreeMap / TreeSet

TreeMapとTreeSetは[け](#)によってキー/をしているためしたがって、TreeMapキーとTreeSetは、いにするがあります。

カスタムPersonクラスがあるとします。

```
public class Person {

    private int id;
    private String firstName, lastName;
    private Date birthday;

    //... Constructors, getters, setters and various methods
}
```

TreeSet または TreeMap の Key にそのまますると

```
TreeSet<Person2> set = ...
set.add(new Person(1, "first", "last", Date.from(Instant.now())));
```

それから、このようにします

```
Exception in thread "main" java.lang.ClassCastException: Person cannot be cast to
java.lang.Comparable
    at java.util.TreeMap.compare(TreeMap.java:1294)
    at java.util.TreeMap.put(TreeMap.java:538)
    at java.util.TreeSet.add(TreeSet.java:255)
```

これをするために、ids private int id のについて Person インスタンスをしたいとしましょう。私たちは2つのいずれかでそれをうことができます

1. 1つのは、 [Comparable インターフェイス](#) をするように Person をすることです。

```
public class Person implements Comparable<Person> {
    private int id;
    private String firstName, lastName;
    private Date birthday;

    //... Constructors, getters, setters and various methods

    @Override
    public int compareTo(Person o) {
        return Integer.compare(this.id, o.id); //Compare by id
    }
}
```

2. もうつのは、 TreeSet に [Comparator](#) をすることです

Java SE 8

```
TreeSet<Person> treeSet = new TreeSet<>((personA, personB) -> Integer.compare(personA.getId(),
personB.getId()));
```

```
TreeSet<Person> treeSet = new TreeSet<>(new Comparator<Person>() {
    @Override
    public int compare(Person personA, Person personB) {
        return Integer.compare(personA.getId(), personB.getId());
    }
});
```

しかし、のアプローチには2つのがあります。

1. インスタンスが `TreeSet` / `TreeMap` されると、けにされるフィールドをしないことがに `TreeMap`。のでは、コレクションにすでにされているの `id` をすると、しないがするがあります。
2. をかつしてすることがです。 [Javadocによると](#)

は、すべての `x` と `y` にして `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))` をするがあります。これは、`y.compareTo(x)` がをスローする、`x.compareTo(y)` がをスローするがあることをします。

は、がであることをしなければならない。 `(x.compareTo(y)>0 && y.compareTo(z)>0)`
`x.compareTo(z)>0` する。

に、は、すべての `z` にして、`x.compareTo(y)==0` が `sgn(x.compareTo(z)) == sgn(y.compareTo(z))` を `x.compareTo(y)==0` ことをするがあります。

TreeMapとTreeSetスレッドの

`TreeMap` と `TreeSet` はスレッドセーフなコレクションではないので、マルチスレッドプログラムでするときはがです。

`TreeMap` と `TreeSet` は、にのスレッドでみんだでもです。したがって、それらがされてのスレッドたとえば、プログラムのにによってみまれ、のスレッドによってみられ、されない、またはロックのではありません。

ただし、のスレッドがにみりまたはしたり、のスレッドによってにされた、コレクションは [ConcurrentModificationException](#) をスローしたり、しないをするがあります。このようなは、のいずれかのをして、コレクションへのアクセスを/ロックすることがです。

1. `Collections.synchronizedSorted..`

```
SortedSet<Integer> set = Collections.synchronizedSortedSet(new TreeSet<Integer>());
SortedMap<Integer,String> map = Collections.synchronizedSortedMap(new
TreeMap<Integer,String>());
```

これにより、のコレクションをにした [SortedSet](#) / [SortedMap](#) がされ、の `mutex` オブジェクトでされます。これは、のロックのコレクションへのすべてのみりおよびきみアクセスをさせるため、みりでさえであることにしてください。

2. コレクションのように、のオブジェクトをですることによって

```
TreeSet<Integer> set = new TreeSet<>();
```

...

```
//Thread 1
```

```
synchronized (set) {  
    set.add(4);  
}
```

...

```
//Thread 2  
synchronized (set) {  
    set.remove(5);  
}
```

3. `ReentrantReadWriteLock`などのロックをすると

```
TreeSet<Integer> set = new TreeSet<>();  
ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
```

...

```
//Thread 1  
lock.writeLock().lock();  
set.add(4);  
lock.writeLock().unlock();
```

...

```
//Thread 2  
lock.readLock().lock();  
set.contains(5);  
lock.readLock().unlock();
```

のとはに、`ReadWriteLock`をすると、のスレッドがマップからにみることができます。

オンラインでTreeMapとTreeSetをむ <https://riptutorial.com/ja/java/topic/9905/treemapとtreeset>

68: XJC

き

XJCは、XMLスキーマファイルをにきのJavaクラスにコンパイルするJava SEツールです。

これはJDKパッケージでされ、 `/bin/xjc` pathにあります。

- `xjc [options]スキーマファイル/ URL / dir / jar ... [-b bindinfo] ...`

パラメーター

パラメータ	
スキーマファイル	javaにするxsdスキーマファイル

XJCツールは、JDKのとしてできます。これにより、unマーシャリングにした**JAXB**アノテーションでかけられたJavaコードをすることができます。

Examples

な**XSD**ファイルから**Java**コードをする

XSDスキーマ schema.xsd

のxmlスキーマxsdは、`name`と`reputation`つユーザーのリストをします。

```
<?xml version="1.0"?>

<xs:schema version="1.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:ns="http://www.stackoverflow.com/users"
  elementFormDefault="qualified"
  targetNamespace="http://www.stackoverflow.com/users">
  <xs:element name="users" type="ns:Users"/>

  <xs:complexType name="Users">
    <xs:sequence>
      <xs:element type="ns:User" name="user" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="User">
    <xs:attribute name="name" use="required" type="xs:string"/>
    <xs:attribute name="reputation" use="required">
```

```

        <xs:simpleType>
            <xs:restriction base="xs:int">
                <xs:minInclusive value="1"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
</xs:complexType>
</xs:schema>

```

xjcの

これには、OSパスにxjcツールJDKバイナリへのパスが必要です。

コードのは、

```
xjc schema.xsd
```

これにより、ディレクトリにjavaファイルがされます。

ファイル

のコメントがいくつかありますが、にされるjavaファイルはのようになります。

```

package com.stackoverflow.users;

import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlType;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Users", propOrder = {
    "user"
})
public class Users {

    protected List<User> user;

    public List<User> getUser() {
        if (user == null) {
            user = new ArrayList<User>();
        }
        return this.user;
    }

}

```

```

package com.stackoverflow.users;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;

```

```

import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlType;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "User")
public class User {

    @XmlAttribute(name = "name", required = true)
    protected String name;
    @XmlAttribute(name = "reputation", required = true)
    protected int reputation;

    public String getName() {
        return name;
    }

    public void setName(String value) {
        this.name = value;
    }

    public int getReputation() {
        return reputation;
    }

    public void setReputation(int value) {
        this.reputation = value;
    }

}

```

```

package com.stackoverflow.users;

import javax.xml.bind.JAXBElement;
import javax.xml.bind.annotation.XmlElementDecl;
import javax.xml.bind.annotation.XmlRegistry;
import javax.xml.namespace.QName;

@XmlRegistry
public class ObjectFactory {

    private final static QName _Users_QNAME = new QName("http://www.stackoverflow.com/users",
"users");

    public ObjectFactory() {
    }

    public Users createUsers() {
        return new Users();
    }

    public User createUser() {
        return new User();
    }

    @XmlElementDecl(namespace = "http://www.stackoverflow.com/users", name = "users")
    public JAXBElement<Users> createUsers(Users value) {
        return new JAXBElement<Users>(_Users_QNAME, Users.class, null, value);
    }

}

```

package-info.java

```
@javax.xml.bind.annotation.XmlSchema(namespace = "http://www.stackoverflow.com/users",  
elementFormDefault = javax.xml.bind.annotation.XmlNsForm.QUALIFIED)  
package com.stackoverflow.users;
```

オンラインでXJCをむ <https://riptutorial.com/ja/java/topic/4538/xjc>

69: XML XPath

XPathは、のまたはノードのなど、XMLツリーの1つのノードをナビゲートしてするためにされます。

このにするとして、 [このW3C](#)をしてください。

Examples

XMLのNodeListの

えられたXML

```
<documentation>
  <tags>
    <tag name="Java">
      <topic name="Regular expressions">
        <example>Matching groups</example>
        <example>Escaping metacharacters</example>
      </topic>
      <topic name="Arrays">
        <example>Looping over arrays</example>
        <example>Converting an array to a list</example>
      </topic>
    </tag>
    <tag name="Android">
      <topic name="Building Android projects">
        <example>Building an Android application using Gradle</example>
        <example>Building an Android application using Maven</example>
      </topic>
      <topic name="Layout resources">
        <example>Including layout resources</example>
        <example>Supporting multiple device screens</example>
      </topic>
    </tag>
  </tags>
</documentation>
```

のexampleは、JavaタグのすべてのノードをしますXMLのXPathのみをするはこのメソッドをします。のXPathびしがじXMLファイルでされるののをしてください。

```
XPathFactory xPathFactory = XPathFactory.newInstance();
XPath xPath = xPathFactory.newXPath(); //Make new XPath
InputSource inputSource = new InputSource("path/to/xml.xml"); //Specify XML file path

NodeList javaExampleNodes = (NodeList)
xPath.evaluate("/documentation/tags/tag[@name='Java']//example", inputSource,
XPathConstants.NODESET); //Evaluate the XPath
...
```

1つのXMLでのXPathをする

XMLのNodeListをするのとじをして、のXPathびしをにうをにします。

えられたXML

```
<documentation>
  <tags>
    <tag name="Java">
      <topic name="Regular expressions">
        <example>Matching groups</example>
        <example>Escaping metacharacters</example>
      </topic>
      <topic name="Arrays">
        <example>Looping over arrays</example>
        <example>Converting an array to a list</example>
      </topic>
    </tag>
    <tag name="Android">
      <topic name="Building Android projects">
        <example>Building an Android application using Gradle</example>
        <example>Building an Android application using Maven</example>
      </topic>
      <topic name="Layout resources">
        <example>Including layout resources</example>
        <example>Supporting multiple device screens</example>
      </topic>
    </tag>
  </tags>
</documentation>
```

これは、XPathをしてのを1つのドキュメントです。

```
XPath xPath = XPathFactory.newInstance().newXPath(); //Make new XPath
DocumentBuilder builder = DocumentBuilderFactory.newInstance();
Document doc = builder.parse(new File("path/to/xml.xml")); //Specify XML file path

NodeList javaExampleNodes = (NodeList)
xPath.evaluate("/documentation/tags/tag[@name='Java']//example", doc, XPathConstants.NODESET);
//Evaluate the XPath
xPath.reset(); //Resets the XPath so it can be used again
NodeList androidExampleNodes = (NodeList)
xPath.evaluate("/documentation/tags/tag[@name='Android']//example", doc,
XPathConstants.NODESET); //Evaluate the XPath

...
```

のXPathをXMLでする

このケースでは、びしがするように、のにコンパイルみのをちたいevaluateていないcompileじを。なはのようになります。

```
XPath xPath = XPathFactory.newInstance().newXPath(); //Make new XPath
XPathExpression exp = xPath.compile("/documentation/tags/tag[@name='Java']//example");
DocumentBuilder builder = DocumentBuilderFactory.newInstance();
Document doc = builder.parse(new File("path/to/xml.xml")); //Specify XML file path

NodeList javaExampleNodes = (NodeList) exp.evaluate(doc, XPathConstants.NODESET); //Evaluate
```

```
the XPath from the already-compiled expression
```

```
NodeList javaExampleNodes2 = (NodeList) exp.evaluate(doc, XPathConstants.NODESET); //Do it again
```

に、を2びしXPathExpression.evaluate()はるかにには、2つのびしよりなりXPath.evaluate()

オンラインでXML XPathをむ <https://riptutorial.com/ja/java/topic/4148/xml-xpath>

70: XOM - XML オブジェクトモデル

Examples

XML ファイルのみみ

XOMでXMLデータをロードするには、`Builder`をして`Document`にみむがあります。

```
Builder builder = new Builder();
Document doc = builder.build(file);
```

xmlファイルののであるルートをするには、`Document` インスタンスで`getRootElement()`をするがあります。

```
Element root = doc.getRootElement();
```

`Element`クラスにはxmlのみみをにするなメソッドがたくさんあります。もなもののいくつかをにします。

- `getChildElements(String name)` - のとしてする`Elements` インスタンスをします
- `getFirstChildElement(String name)` - そのタグをつのをします。
- `getValue()` - のをします。
- `getAttributeValue(String name)` - されたをつのをします。

`getChildElements()`をびすと、`Elements` インスタンスがされます。これから、ループして`get(int index)`メソッドをびして、のすべてのをできます。

```
Elements colors = root.getChildElements("color");
for (int q = 0; q < colors.size(); q++){
    Element color = colors.get(q);
}
```

XMLファイルを見るをにします。

XMLファイル

```

1  <example>
2      <person>
3          <name>
4              <first>Dan</first>
5              <last>Smith</last>
6          </name>
7          <age unit="years">23</age>
8          <fav_color>green</fav_color>
9      </person>
10     <person>
11         <name>
12             <first>Bob</first>
13             <last>Autry</last>
14         </name>
15         <age unit="months">3</age>
16         <fav_color>N/A</fav_color>
17     </person>
18 </example>

```

それを見、するためのコード

```

import java.io.File;
import java.io.IOException;
import nu.xom.Builder;
import nu.xom.Document;
import nu.xom.Element;
import nu.xom.Elements;
import nu.xom.ParsingException;

public class XMLReader {

    public static void main(String[] args) throws ParsingException, IOException{
        File file = new File("insert path here");
        // builder builds xml data
        Builder builder = new Builder();
        Document doc = builder.build(file);

        // get the root element <example>
        Element root = doc.getRootElement();

        // gets all element with tag <person>
        Elements people = root.getChildElements("person");

        for (int q = 0; q < people.size(); q++){
            // get the current person element
            Element person = people.get(q);

            // get the name element and its children: first and last
            Element nameElement = person.getFirstChildElement("name");
            Element firstNameElement = nameElement.getFirstChildElement("first");
            Element lastNameElement = nameElement.getFirstChildElement("last");

            // get the age element

```

```

        Element ageElement = person.getFirstChildElement("age");

        // get the favorite color element
        Element favColorElement = person.getFirstChildElement("fav_color");

        String fName, lName, ageUnit, favColor;
        int age;

        try {
            fName = firstNameElement.getValue();
            lName = lastNameElement.getValue();
            age = Integer.parseInt(ageElement.getValue());
            ageUnit = ageElement.getAttributeValue("unit");
            favColor = favColorElement.getValue();

            System.out.println("Name: " + lName + ", " + fName);
            System.out.println("Age: " + age + " (" + ageUnit + ")");
            System.out.println("Favorite Color: " + favColor);
            System.out.println("-----");

        } catch (NullPointerException ex){
            ex.printStackTrace();
        } catch (NumberFormatException ex){
            ex.printStackTrace();
        }
    }
}

```

これはコンソールにされます

```

Name: Smith, Dan
Age: 23 (years)
Favorite Color: green
-----
Name: Autry, Bob
Age: 3 (months)
Favorite Color: N/A
-----

```

XML ファイルへのきみ

XOMをしてXMLファイルにきむことは、このはインスタンスをrootからするのではなく、インスタンスをしているをいて、みりとよくています。

しいElementをするには、コンストラクター`Element(String name)`ます。ルートをして、`Document`にすることができます。

```

Element root = new Element("root");

```

`Element`クラスには、をするためのなメソッドがいくつかあります。それらはのりです

- `appendChild(String name)` - これはにのをnameにします。

- `appendChild(Node node)` - `node`のをにします。はノードなので、をできます。
- `addAttribute(Attribute attribute)` - にをします。

`Attribute`クラスには、いくつかのなるコンストラクタがあります。もなのは`Attribute(String name, String value)`です。

すべてのをルートにしたら、それを`Document`にできます。`Document`は、そのコンストラクタのとして`Element`をとります。

`Serializer`をしてXMLをファイルにきむことができます。`Serializer`のコンストラクタであるためにしいストリームをするがあります。

```
FileOutputStream fileOutputStream = new FileOutputStream(file);
Serializer serializer = new Serializer(fileOutputStream, "UTF-8");
serializer.setIndent(4);
serializer.write(doc);
```

コード

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.UnsupportedEncodingException;
import nu.xom.Attribute;
import nu.xom.Builder;
import nu.xom.Document;
import nu.xom.Element;
import nu.xom.Elements;
import nu.xom.ParsingException;
import nu.xom.Serializer;

public class XMLWriter{

    public static void main(String[] args) throws UnsupportedEncodingException,
        IOException{
        // root element <example>
        Element root = new Element("example");

        // make a array of people to store
        Person[] people = {new Person("Smith", "Dan", "years", "green", 23),
            new Person("Autry", "Bob", "months", "N/A", 3)};

        // add all the people
        for (Person person : people){

            // make the main person element <person>
            Element personElement = new Element("person");

            // make the name element and it's children: first and last
            Element nameElement = new Element("name");
            Element firstNameElement = new Element("first");
            Element lastNameElement = new Element("last");

            // make age element
```

```

        Element ageElement = new Element("age");

        // make favorite color element
        Element favColorElement = new Element("fav_color");

        // add value to names
        firstNameElement.appendChild(person.getFirstName());
        lastNameElement.appendChild(person.getLastName());

        // add names to name
        nameElement.appendChild(firstNameElement);
        nameElement.appendChild(lastNameElement);

        // add value to age
        ageElement.appendChild(String.valueOf(person.getAge()));

        // add unit attribute to age
        ageElement.addAttribute(new Attribute("unit", person.getAgeUnit()));

        // add value to favColor
        favColorElement.appendChild(person.getFavoriteColor());

        // add all contents to person
        personElement.appendChild(nameElement);
        personElement.appendChild(ageElement);
        personElement.appendChild(favColorElement);

        // add person to root
        root.appendChild(personElement);
    }

    // create doc off of root
    Document doc = new Document(root);

    // the file it will be stored in
    File file = new File("out.xml");
    if (!file.exists()){
        file.createNewFile();
    }

    // get a file output stream ready
    FileOutputStream fileOutputStream = new FileOutputStream(file);

    // use the serializer class to write it all
    Serializer serializer = new Serializer(fileOutputStream, "UTF-8");
    serializer.setIndent(4);
    serializer.write(doc);
}

private static class Person {

    private String lName, fName, ageUnit, favColor;
    private int age;

    public Person(String lName, String fName, String ageUnit, String favColor, int age){
        this.lName = lName;
        this.fName = fName;
        this.age = age;
        this.ageUnit = ageUnit;
        this.favColor = favColor;
    }
}

```



```

    public String getLastName() { return lName; }
    public String getFirstName() { return fName; }
    public String getAgeUnit() { return ageUnit; }
    public String getFavoriteColor() { return favColor; }
    public int getAge() { return age; }
}
}

```

これは "out.xml" のになります。

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <example>
3      <person>
4          <name>
5              <first>Dan</first>
6              <last>Smith</last>
7          </name>
8          <age unit="years">23</age>
9          <fav_color>green</fav_color>
10     </person>
11     <person>
12         <name>
13             <first>Bob</first>
14             <last>Autry</last>
15         </name>
16         <age unit="months">3</age>
17         <fav_color>N/A</fav_color>
18     </person>
19 </example>
20

```

オンラインでXOM - XMLオブジェクトモデルをむ [https://riptutorial.com/ja/java/topic/5091/xom----](https://riptutorial.com/ja/java/topic/5091/xom----xmlオブジェクトモデル)
-xmlオブジェクトモデル

71: アサーション

- `assert expression1 ;`
- `assert expression1 expression2 ;`

パラメーター

パラメーター	
1	このが <code>false</code> されると、 <code>assertion</code> ステートメントは <code>AssertionError</code> をスローし <code>false</code> 。
2	オプション。すると、 <code>assert</code> によってスローされた <code>AssertionError</code> はこのメッセージをちます。

デフォルトでは、アサーションはにになります。

アサーションをにするには、`-ea` フラグをして`java`をするがあり`-ea`。

```
java -ea com.example.AssertionExample
```

アサーションは、が`false`されたにエラーをスローするステートメントです。アサーションはコードのテストにのみするがあります。プロダクションでしてしないでください。

Examples

アサートによるのチェック

```
a = 1 - Math.abs(1 - a % 2);

// This will throw an error if my arithmetic above is wrong.
assert a >= 0 && a <= 1 : "Calculated value of " + a + " is outside of expected bounds";

return a;
```

オンラインでアサーションをむ <https://riptutorial.com/ja/java/topic/407/アサーション>

72: アプレット

き

アプレットはリリースJavaのであり、もJavaとプログラミングをえるためにされてきました。

、アプレットやのブラウザプラグインから、ブラウザをブロックしたりにサポートしていないブラウザプラグインからのがにんでいます。

2016に、オラクルは、プラグインをするをしました。[プラグインフリーWebへの](#)

よりしい、よりいAPIがになりました

アプレットは、はWebブラウザでされるJavaアプリケーションです。なえは、サーバーとしてをするなく、ユーザーとすることです。このコンセプトは、インターネットがくてだった2000ににしました。

アプレットはライフサイクルをする5つのメソッドをします。

メソッド	
<code>init()</code>	アプレットがみまれるとびされます
<code>destroy()</code>	アプレットがメモリからされるとびされます
<code>start()</code>	アプレットがになるたびにびされます
<code>stop()</code>	アプレットがのウィンドウによってオーバーラップするたびにびされます
<code>paint()</code>	にじてびされるか、または <code>repaint()</code> びすことによってでトリガされます。

Examples

アプレット

にシンプルなアプレットはをき、をスクリーンにします。

```
public class MyApplet extends JApplet{

    private String str = "StackOverflow";

    @Override
    public void init() {
        setBackground(Color.gray);
    }
    @Override
```

```

public void destroy() {}
@Override
public void start() {}
@Override
public void stop() {}
@Override
public void paint(Graphics g) {
    g.setColor(Color.yellow);
    g.fillRect(1,1,300,150);
    g.setColor(Color.red);
    g.setFont(new Font("TimesRoman", Font.PLAIN, 48));
    g.drawString(str, 10, 80);
}
}

```

アプレットのメインクラスは `javax.swing.JApplet` から `javax.swing.JApplet` ます。

Java SE 1.2

Java 1.2では、Swapp APIアプレットのは `java.applet.Applet` からされました。

アプレットはmainメソッドをとしません。エントリポイントは、ライフサイクルによってされます。それらをするには、HTMLにめむがあります。これは、そのサイズがされているポイントでもあります。

```

<html>
  <head></head>
  <body>
    <applet code="MyApplet.class" width="400" height="200"></applet>
  </body>
</html>

```

GUIの

アプレットをしてGUIをにできます。らは `Container` ようにし、 `awt` または `swing` コンポーネントをとる `add()` メソッドをとっています。

```

public class MyGUIApplet extends JApplet{

    private JPanel panel;
    private JButton button;
    private JComboBox<String> cmbBox;
    private JTextField textField;

    @Override
    public void init(){
        panel = new JPanel();
        button = new JButton("ClickMe!");
        button.addActionListener(new ActionListener(){
            @Override
            public void actionPerformed(ActionEvent ae) {
                if(((String)cmbBox.getSelectedItem()).equals("greet")) {
                    JOptionPane.showMessageDialog(null,"Hello " + textField.getText());
                } else {
                    JOptionPane.showMessageDialog(null,textField.getText() + " stinks!");
                }
            }
        });
    }
}

```

```

        }
    }
    });
    cmbBox = new JComboBox<>(new String[]{"greet", "offend"});
    textField = new JTextField("John Doe");
    panel.add(cmbBox);
    panel.add(textField);
    panel.add(button);
    add(panel);
}
}

```

アプレットからのリンクをく

メソッド `getAppletContext()` をして `AppletContext` オブジェクトをし、ブラウザにリンクをくようにすることができます。このためには、`showDocument()` メソッドをします。2のパラメータは、ブラウザにしいウィンドウ `_blank` をするように、またはアプレット `_self` をするようにブラウザにし `_self`。

```

public class MyLinkApplet extends JApplet{
    @Override
    public void init(){
        JButton button = new JButton("ClickMe!");
        button.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent ae) {
                AppletContext a = getAppletContext();
                try {
                    URL url = new URL("http://stackoverflow.com/");
                    a.showDocument(url, "_blank");
                } catch (Exception e) { /* omitted for brevity */ }
            }
        });
        add(button);
    }
}

```

、オーディオ、そののリソースのみみ

Javaアプレットはなるリソースをみむことができます。しかし、らはクライアントのWebブラウザでされているので、これらのリソースにアクセスであることをするがあります。アプレットはローカルファイルシステムとしてクライアントリソースにアクセスできません。

アプレットがされているURLからリソースをロードするは、メソッド `getCodeBase()` をしてベースURLをできます。リソースをロードするために、アプレットは `getImage()` メソッドと `getAudioClip()` メソッドをしてやオーディオファイルをロードします。

をみんでする

```

public class MyImgApplet extends JApplet{

    private Image img;

    @Override
    public void init(){
        try {
            img = getImage(new URL("http://cdn.sstatic.net/stackexchange/img/logos/so/so-
logo.png"));
        } catch (MalformedURLException e) { /* omitted for brevity */ }
    }
    @Override
    public void paint(Graphics g) {
        g.drawImage(img, 0, 0, this);
    }
}

```

オーディオファイルをみんでする

```

public class MyAudioApplet extends JApplet{

    private AudioClip audioClip;

    @Override
    public void init(){
        try {
            audioClip = getAudioClip(new URL("URL/TO/AN/AUDIO/FILE.WAV"));
        } catch (MalformedURLException e) { /* omitted for brevity */ }
    }
    @Override
    public void start() {
        audioClip.play();
    }
    @Override
    public void stop(){
        audioClip.stop();
    }
}

```

テキストファイルをみんでする

```

public class MyTextApplet extends JApplet{
    @Override
    public void init(){
        JTextArea textArea = new JTextArea();
        JScrollPane sp = new JScrollPane(textArea);
        add(sp);
        // load text
        try {
            URL url = new URL("http://www.textfiles.com/fun/quotes.txt");
            InputStream in = url.openStream();
            BufferedReader bf = new BufferedReader(new InputStreamReader(in));

```

```
String line = "";
while((line = bf.readLine()) != null) {
    textArea.append(line + "\n");
}
} catch(Exception e) { /* omitted for brevity */ }
}
```

オンラインでアプレットをむ <https://riptutorial.com/ja/java/topic/5503/アプレット>

73: イテレータとイテレータ

き

`java.util.Iterator`は、`Iterator`デザインパターンをするオブジェクトのJava SEインターフェイスです。 `java.lang.Iterable` インタフェースは、イテレータをできるオブジェクトです。

Javaは`Iterable`をしません、`for-each`ループをしてをすることはです。は、バックグラウンドでアクセスできないインデックスをしてJVMによってわれます。

Examples

forループでIterableをう

`Iterable<>`インタフェースをするクラスは、`for`ループでできます。これは、オブジェクトからイテレータをし、それをしてすべてのをにするためのです。コードをよりにし、みをくし、エラーをこしにくくします。

```
public class UsingIterable {

    public static void main(String[] args) {
        List<Integer> intList = Arrays.asList(1,2,3,4,5,6,7);

        // List extends Collection, Collection extends Iterable
        Iterable<Integer> iterable = intList;

        // foreach-like loop
        for (Integer i: iterable) {
            System.out.println(i);
        }

        // pre java 5 way of iterating loops
        for(Iterator<Integer> i = iterable.iterator(); i.hasNext(); ) {
            Integer item = i.next();
            System.out.println(item);
        }
    }
}
```

のイテレータをする

`foreach`ループまたは "extended for loop"をするのはですが、イテレータをするとなることがあります。たとえば、コンマりのをしたいが、のにコンマをつけたくないは、のようにします。

```
List<String> yourData = //...
Iterator<String> iterator = yourData.iterator();
while (iterator.hasNext()){
    // next() "moves" the iterator to the next entry and returns it's value.
```



```
String entry = iterator.next();
System.out.print(entry);
if (iterator.hasNext()){
    // If the iterator has another element after the current one:
    System.out.print(",");
}
}
```

`isLastEntry`をするか、ループインデックスをしてするよりもはるかにでかりやすくなります。

の**Iterable**をする。

どのインタフェースでもの**Iterable**をするには、インタフェースにメソッドをするだけです。

Iterableには`iterator()`とばれるものが1つしかありません。しかし、そのリターン `Iterator` はそれ、3つのメソッドをつインタフェースです。コレクションにけられたイテレータをすことも、のカスタムをすることもできます。

```
public static class Alphabet implements Iterable<Character> {

    @Override
    public Iterator<Character> iterator() {
        return new Iterator<Character>() {
            char letter = 'a';

            @Override
            public boolean hasNext() {
                return letter <= 'z';
            }

            @Override
            public Character next() {
                return letter++;
            }

            @Override
            public void remove() {
                throw new UnsupportedOperationException("Doesn't make sense to remove a
letter");
            }
        };
    }
}
```

するには

```
public static void main(String[] args) {
    for(char c : new Alphabet()) {
        System.out.println("c = " + c);
    }
}
```

しい `Iterator` は、のアイテムをしすでなければなりません `Iterator` へのびしは、のをすようにをします。 `hasNext()` はイテレータがあるかどうかをチェックします。イテレータがなコレクションにされている、イテレータのオプションの `remove()` メソッドがされ、のコレクションからポイン

トされているアイテムをできます。

イテレータをしたの

`Iterator.remove()` メソッドは、`Iterator.next()` へののびしによってされたをするオプションのメソッドです。たとえば、のコードはのリストをし、すべてののをします。

```
List<String> names = new ArrayList<>();
names.add("name 1");
names.add("name 2");
names.add("");
names.add("name 3");
names.add("");
System.out.println("Old Size : " + names.size());
Iterator<String> it = names.iterator();
while (it.hasNext()) {
    String el = it.next();
    if (el.equals("")) {
        it.remove();
    }
}
System.out.println("New Size : " + names.size());
```

```
Old Size : 5
New Size : 3
```

のコードは、なコレクションをしながらをするなです。わりに、のようにコレクションからをしようとします。

```
for (String el: names) {
    if (el.equals("")) {
        names.remove(el); // WRONG!
    }
}
```

イテレータにフェイル・ファスト・イテレータ・セマンティクスをするなコレクション `ArrayList` など
は、`ConcurrentModificationException` をスローします。

`remove()` メソッドは、`next()` びしのにのみびされます。`next()` びすにびされた、または`next()` びし
のに2びされた、`remove()` びしは`IllegalStateException` をスローします。

`remove` はオプションのとしてされています。つまり、すべてのイテレータでされるわけではありません。
サポートされていないには、なコレクションのイテレータ、コレクションのみのビュー、
サイズのコレクションなどがあります。イテレータがをサポートしていないときに
`remove()` がびされると、`UnsupportedOperationException` がスローされます。

オンラインでイテレータとイテレータをむ <https://riptutorial.com/ja/java/topic/172/イテレータとイテレータ>

74: インターフェイス

き

インタフェースはクラスとので、`interface` キーワードをしてできます。インターフェイスには、メソッドシグネチャ、デフォルトメソッド、メソッド、およびネストされたのみをめることができます。メソッドは、デフォルトメソッドとメソッドにしてのみします。クラスのように、インタフェースはインスタンスすることができません。インタフェースはクラスによってのみでき、のインタフェースですることもできます。インタフェースは、Javaでなをするためのなです。

- パブリックインターフェイス `Foo {void foo; /*のメソッド*/ }`
- パブリックインターフェイス `Foo1` は `Foo {void bar; /*のメソッド*/ }`
- `public class Foo2` は `Foo`、`Foo1` をしています `{/* Foo と Foo1 の*/ }`

Examples

インターフェイスのと

`interface` キーワードをした `interface`

```
public interface Animal {
    String getSound(); // Interface methods are public by default
}
```

をきする

```
@Override
public String getSound() {
    // Code goes here...
}
```

これにより、コンパイラはオーバーライドしていることをし、プログラムがしいメソッドをしたり、メソッドのシグネチャをとってしまうことをぎます。

インタフェースは、`implements` キーワード をして `implements` ます。

```
public class Cat implements Animal {

    @Override
    public String getSound() {
        return "meow";
    }
}

public class Dog implements Animal {

    @Override
```

```

    public String getSound() {
        return "woof";
    }
}

```

ここでは、インターフェイスのメソッドがにメソッドであるため、クラス `Cat` および `Dog` は `getSound()` メソッドをするがあります `getSound()` デフォルトメソッドをく。

インタフェースの

```

Animal cat = new Cat();
Animal dog = new Dog();

System.out.println(cat.getSound()); // prints "meow"
System.out.println(dog.getSound()); // prints "woof"

```

のインタフェースの

Javaクラスは、のインタフェースをできます。

```

public interface NoiseMaker {
    String noise = "Making Noise"; // interface variables are public static final by default

    String makeNoise(); //interface methods are public abstract by default
}

public interface FoodEater {
    void eat(Food food);
}

public class Cat implements NoiseMaker, FoodEater {
    @Override
    public String makeNoise() {
        return "meow";
    }

    @Override
    public void eat(Food food) {
        System.out.println("meows appreciatively");
    }
}

```

`Cat` クラスがのインタフェースでした `abstract` メソッドをするがあることにしてください。さらに、クラスがになのインタフェースをにできるにしてください [JVMの](#)ために **65,535** というがあります。

```

NoiseMaker noiseMaker = new Cat(); // Valid
FoodEater foodEater = new Cat(); // Valid
Cat cat = new Cat(); // valid

Cat invalid1 = new NoiseMaker(); // Invalid
Cat invalid2 = new FoodEater(); // Invalid

```

1. インタフェースでされたすべてののは `public static final`
2. インタフェースのメソッドでされたすべてのメソッドがされている `public abstract` こののはJava 8からのJava 7をしてである、あなたはであるはないインタフェースのメソッドを、つことがされている。このようなは、のようになられている **デフォルトの**
3. インタフェースを `final` としてすることはできません
4. のインタフェースがのシグネチャをつメソッドをしている、に1つのメソッドとしてわれ、どのインタフェースメソッドがされているかをできません
5. する **InterfaceName.class** ファイルは、コンパイルにインタフェースにしてされます

インタフェースの

インタフェースは、 `extends` キーワードによってのインタフェースをすることができます。

```
public interface BasicResourceService {
    Resource getResource();
}

public interface ExtendedResourceService extends BasicResourceService {
    void updateResource(Resource resource);
}
```

これで、 `ExtendedResourceService` をするクラスは、 `getResource()` と `updateResource()` をするがあります。

のインターフェースをする

クラスとはなり、 `extends` キーワードをして、のインタフェースをカンマでってすることができ、インタフェースをしいインタフェースにみわせることができます

```
public interface BasicResourceService {
    Resource getResource();
}

public interface AlternateResourceService {
    Resource getAlternateResource();
}

public interface ExtendedResourceService extends BasicResourceService,
AlternateResourceService {
    Resource updateResource(Resource resource);
}
```

この、 `ExtendedResourceService` をするクラスは、 `getResource()` 、 `getAlternateResource()` 、 および `updateResource()` をするがあります。

Genericsでのインタフェースの

たとえば、AMQP、JMSなどのさまざまなチャンネルとのでデータをパブリッシュ/するインターフェイスをしたいとしますが、のをりえることができるようにしたいとしましょう。

のでできるなIOインターフェイスをしましょう

```
public interface IO<IncomingType, OutgoingType> {  
  
    void publish(OutgoingType data);  
    IncomingType consume();  
    IncomingType RPCSubmit(OutgoingType data);  
  
}
```

はそのインタフェースをインスタンスすることができますが、これらのメソッドのデフォルトのがないので、インスタンスするにはがです。

```
IO<String, String> mockIO = new IO<String, String>() {  
  
    private String channel = "somechannel";  
  
    @Override  
    public void publish(String data) {  
        System.out.println("Publishing " + data + " to " + channel);  
    }  
  
    @Override  
    public String consume() {  
        System.out.println("Consuming from " + channel);  
        return "some useful data";  
    }  
  
    @Override  
    public String RPCSubmit(String data) {  
        return "received " + data + " just now ";  
    }  
  
};  
  
mockIO.consume(); // prints: Consuming from somechannel  
mockIO.publish("TestData"); // Publishing TestData to somechannel  
System.out.println(mockIO.RPCSubmit("TestData")); // received TestData just now
```

たちはまた、そのインタフェースでもっとなことをすることができます。なRabbitMQをラップするためにいたいとしましょう

```
public class RabbitMQ implements IO<String, String> {  
  
    private String exchange;  
    private String queue;  
  
    public RabbitMQ(String exchange, String queue){  
        this.exchange = exchange;  
        this.queue = queue;  
    }  
  
    @Override  
    public void publish(String data) {  
        rabbit.basicPublish(exchange, queue, data.getBytes());  
    }  
  
}
```

```

@Override
public String consume() {
    return rabbit.basicConsume(exchange, queue);
}

@Override
public String RPCSubmit(String data) {
    return rabbit.rpcPublish(exchange, queue, data);
}
}

```

にシステムをしてからのウェブサイトへののをえ、をできるようにするためにこのIOインターフェイスをしたいと思います。

```

import java.util.concurrent.atomic.AtomicLong;

public class VisitCounter implements IO<Long, Integer> {

    private static AtomicLong websiteCounter = new AtomicLong(0);

    @Override
    public void publish(Integer count) {
        websiteCounter.addAndGet(count);
    }

    @Override
    public Long consume() {
        return websiteCounter.get();
    }

    @Override
    public Long RPCSubmit(Integer count) {
        return websiteCounter.addAndGet(count);
    }
}

```

VisitCounterをしましょう

```

VisitCounter counter = new VisitCounter();

// just had 4 visits, yay
counter.publish(4);
// just had another visit, yay
counter.publish(1);

// get data for stats counter
System.out.println(counter.consume()); // prints 5

// show data for stats counter page, but include that as a page view
System.out.println(counter.RPCSubmit(1)); // prints 6

```

のインターフェイスをする、じインターフェイスを2することはできません。これはインターフェースにもてはまります。したがって、のコードはで、コンパイルエラーがします。

```
interface Printer<T> {
    void print(T value);
}

// Invalid!
class SystemPrinter implements Printer<Double>, Printer<Integer> {
    @Override public void print(Double d){ System.out.println("Decimal: " + d); }
    @Override public void print(Integer i){ System.out.println("Discrete: " + i); }
}
```

インタフェースの

くの、インタフェースはにです。たとえば、のリストがあり、リストをループして、それぞれのサウンドをしたいとします。

```
{cat, dog, bird}
```

これをう1つの、インターフェイスをすることです。これにより、すべてのクラスでメソッドをびすことができます

```
public interface Animal {
    public String getSound();
}
```

implements Animalをimplements Animalクラスでも、getSound()メソッドがですが、それらはすべてなるをつことができます

```
public class Dog implements Animal {
    public String getSound() {
        return "Woof";
    }
}

public class Cat implements Animal {
    public String getSound() {
        return "Meow";
    }
}

public class Bird implements Animal{
    public String getSound() {
        return "Chirp";
    }
}
```

3つのクラスがあり、それぞれにgetSound()メソッドがあります。これらのクラスのすべてがあるのでimplement Animalするインターフェイス、getSound()メソッドを、のインスタンスAnimalつことができますgetSound()それにびかけ

```
Animal dog = new Dog();
Animal cat = new Cat();
Animal bird = new Bird();
```



```
dog.getSound(); // "Woof"
cat.getSound(); // "Meow"
bird.getSound(); // "Chirp"
```

これらのそれぞれがあるので、`Animal`、たちも、それらをして、リストのループを、をれて、そのをプリントアウトすることができ

```
Animal[] animals = { new Dog(), new Cat(), new Bird() };
for (Animal animal : animals) {
    System.out.println(animal.getSound());
}
```

のは `Dog`、`Cat`、`Bird` のであるため、「Woof Meow Chirp」がコンソールにされます。

インタフェースは、のりとしてもできます。たとえば、が「dog」のは `Dog` し、が「cat」のは `Cat`、
「bird」のは `Bird` し、そののサウンドをするには、

```
public Animal getAnimalByName(String name) {
    switch(name.toLowerCase()) {
        case "dog":
            return new Dog();
        case "cat":
            return new Cat();
        case "bird":
            return new Bird();
        default:
            return null;
    }
}

public String getAnimalSoundByName(String name){
    Animal animal = getAnimalByName(name);
    if (animal == null) {
        return null;
    } else {
        return animal.getSound();
    }
}

String dogSound = getAnimalSoundByName("dog"); // "Woof"
String catSound = getAnimalSoundByName("cat"); // "Meow"
String birdSound = getAnimalSoundByName("bird"); // "Chirp"
String lightbulbSound = getAnimalSoundByName("lightbulb"); // null
```

しいタイプの `Animal` をしたいは、するでかをするがないため、インターフェイスはにもちます。

クラスでのインタフェースの

`interface` されるメソッドは、デフォルトで `public abstract` です。ときに `abstract class` `interface`、
でされているいづれかの `interface` であるはありません `abstract class`。これは、`abstract` とされ
た `class` メソッドがまれるためです。したがって、のインタフェースおよび/または `abstract class`
からした `abstract` メソッドをするのは、のなサブクラスのです。

```

public interface NoiseMaker {
    void makeNoise();
}

public abstract class Animal implements NoiseMaker {
    //Does not need to declare or implement makeNoise()
    public abstract void eat();
}

//Because Dog is concrete, it must define both makeNoise() and eat()
public class Dog extends Animal {
    @Override
    public void makeNoise() {
        System.out.println("Borf borf");
    }

    @Override
    public void eat() {
        System.out.println("Dog eats some kibble.");
    }
}

```

Java 8では、`interface`がメソッドの`abstract`メソッドではないことをするメソッドの`default`をすることがです。したがって、なサブクラスはメソッドをするはありませんが、オーバーライドされないり`default`をし`default`。

デフォルトのメソッド

Java 8でされたデフォルトメソッドは、インタフェースのをするです。これは、インタフェースのをし、サブクラスをすることによって、な "Base"または "Abstract"クラスをけるためにできます。

オブザーバーパターンの

たとえば、Observer-Listenerパターンをインターフェイスにして、するクラスにをたせることができます。

```

interface Observer {
    void onAction(String a);
}

interface Observable{
    public abstract List<Observer> getObservers();

    public default void addObserver(Observer o){
        getObservers().add(o);
    }

    public default void notify(String something ){
        for( Observer l : getObservers() ){
            l.onAction(something);
        }
    }
}

```

```
}
```

ここでObservableインターフェースをすることで、どのクラスも「Observable」にできますが、のクラスとしてのにえます。

```
abstract class Worker{
    public abstract void work();
}

public class MyWorker extends Worker implements Observable {

    private List<Observer> myObservers = new ArrayList<Observer>();

    @Override
    public List<Observer> getObservers() {
        return myObservers;
    }

    @Override
    public void work(){
        notify("Started work");

        // Code goes here...

        notify("Completed work");
    }

    public static void main(String[] args) {
        MyWorker w = new MyWorker();

        w.addListener(new Observer() {
            @Override
            public void onAction(String a) {
                System.out.println(a + " (" + new Date() + ")");
            }
        });

        w.work();
    }
}
```

ダイヤモンドの

Java 8のコンパイラは、クラスがシグネチャをつメソッドをむインタフェースをしているときにする**ダイヤモンド**をしています。

これをするには、クラスがメソッドをオーバーライドして、のをするがあります。

```
interface InterfaceA {
    public default String getName(){
        return "a";
    }
}
```

```

interface InterfaceB {
    public default String getName(){
        return "b";
    }
}

public class ImpClass implements InterfaceA, InterfaceB {

    @Override
    public String getName() {
        //Must provide its own implementation
        return InterfaceA.super.getName() + InterfaceB.super.getName();
    }

    public static void main(String[] args) {
        ImpClass c = new ImpClass();

        System.out.println( c.getName() );           // Prints "ab"
        System.out.println( ((InterfaceA)c).getName() ); // Prints "ab"
        System.out.println( ((InterfaceB)c).getName() ); // Prints "ab"
    }
}

```

コンパイルされないなるりのをつじとパラメータをつメソッドをつことにはとしてがあります。

のをするためののメソッドの

のメソッドは、インターフェイスがいくつかのクラスによってされるのシステムのインターフェイスにメソッドがされたににです。

システムののをけるために、メソッドをインタフェースにするときに、デフォルトのメソッドをすることができます。このでは、システムはまだコンパイルされ、のにはにできます。

については、「[デフォルトメソッド](#)」のトピックをしてください。

インタフェースの

Oracle Javaスタイル・ガイドには、

はのうちにきされるべきではありません。

コンテキストおよびのOracleドキュメントへのリンクは、[Oracle Official Code Standard](#)のをしてください。

このスタイルガイダンスは、にインターフェイスにされます。のコードスニペットをえてみましょう

```

interface I {
    public static final int VARIABLE = 0;
}

```

```
public abstract void method();

public static void staticMethod() { ... }
public default void defaultMethod() { ... }
}
```

すべてのインタフェースは、の `public` すべてアクセス、 `static` インタフェースでアクセス、 `final` にするがありますをつのです。

```
public static final int VARIABLE = 0;
```

メソッド

1. をしないすべてのメソッドは、のうちに `public` で `abstract` です。

```
public abstract void method();
```

Java SE 8

2. `static` または `default` をつすべてのメソッドは、を `public` するがあります。

```
public static void staticMethod() { ... }
```

のがすべてされたら、のようになります

```
interface I {
    int VARIABLE = 0;

    void method();

    static void staticMethod() { ... }
    default void defaultMethod() { ... }
}
```

パラメータをする

きの **パラメーター** をすると、のきにをできます。

```
class SomeClass {
}

class Demo<T extends SomeClass> {
}
```

しかし、パラメータはのクラスにしかできません。

インターフェイスタイプは、すでにバインディングを行っているタイプにバインドできます。これは、`&`をしてされます。

```
interface SomeInterface {  
  
}  
  
class GenericClass<T extends SomeClass & SomeInterface> {  
  
}
```

これによりバインドがされ、のからするがになるがあります。

のインターフェイスタイプをタイプパラメータにバインドすることができます。

```
class Demo<T extends SomeClass & FirstInterface & SecondInterface> {  
  
}
```

しかし、してするがあります。のインタフェースバインディングは、、[コードのいの](#)であり、のタイプのアダプタとしてするしいタイプをするがあることをしています。

```
interface NewInterface extends FirstInterface, SecondInterface {  
  
}  
  
class Demo<T extends SomeClass & NewInterface> {  
  
}
```

オンラインでインターフェイスをむ <https://riptutorial.com/ja/java/topic/102/インターフェイス>

75: オーディオ

`javax.sound.sampled.Clip`をするわりに、アプレットAPIからの`AudioClip`をすることもできます。ただし、`AudioClip`はくたがられているため、`Clip`をすることをおめします。

Examples

ループされたオーディオファイルをする

な

```
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.Clip;
```

このコードではクリップがされ、されるとしてされます

```
Clip clip = AudioSystem.getClip();
clip.open(AudioSystem.getAudioInputStream(new URL(filename)));
clip.start();
clip.loop(Clip.LOOP_CONTINUOUSLY);
```

サポートされているすべてのファイルタイプのをしします。

```
AudioFileFormat.Type [] audioFileTypes = AudioSystem.getAudioFileTypes();
```

MIDIファイルをする

MIDIファイルは、`javax.sound.midi`パッケージののクラスをしてできます。`Sequencer`はMIDIファイルのをい、そのくのをしてループカウント、テンポ、トラックミュートなどのコントロールをできます。

このようにして、MIDIデータのなをうことができます。

```
import java.io.File;
import java.io.IOException;
import javax.sound.midi.InvalidMidiDataException;
import javax.sound.midi.MidiSystem;
import javax.sound.midi.MidiUnavailableException;
import javax.sound.midi.Sequence;
import javax.sound.midi.Sequencer;

public class MidiPlayback {
    public static void main(String[] args) {
        try {
            Sequencer sequencer = MidiSystem.getSequencer(); // Get the default Sequencer
            if (sequencer==null) {
                System.err.println("Sequencer device not supported");
                return;
            }
        }
    }
}
```

```

    }
    sequencer.open(); // Open device
    // Create sequence, the File must contain MIDI file data.
    Sequence sequence = MidiSystem.getSequence(new File(args[0]));
    sequencer.setSequence(sequence); // load it into sequencer
    sequencer.start(); // start the playback
} catch (MidiUnavailableException | InvalidMidiDataException | IOException ex) {
    ex.printStackTrace();
}
}
}

```

をするには

```
sequencer.stop(); // Stop the playback
```

シーケンサーは、にシーケンスのトラックの1つまたはをミュートするようにすることができ、されたはされません。のでは、ミュートするシーケンスののトラックをします。

```

import javax.sound.midi.Track;
// ...

Track[] track = sequence.getTracks();
sequencer.setTrackMute(track[0]);

```

ループカウントがえられた、シーケンサはシーケンスをりしできます。のでは、シーケンスを4、にするようにシーケンサーをしています。

```

sequencer.setLoopCount(3);
sequencer.setLoopCount(Sequencer.LOOP_CONTINUOUSLY);

```

シーケンサは、ずしもからシーケンスをするはなく、までシーケンスをするありません。これは、してし、でするでダニをすることで、のすることができます。シーケンサーがするシーケンスのティックをですることもできます。

```

sequencer.setLoopStartPoint(512);
sequencer.setLoopEndPoint(32768);
sequencer.setTickPosition(8192);

```

シーケンサーは、のテンポでMIDIファイルをすることもできます。これは、テンポを1あたりのBPMまたは4MPQあたりのマイクロですることできます。シーケンスがされるものにすることができます。

```

sequencer.setTempoInBPM(1250f);
sequencer.setTempoInMPQ(4750f);
sequencer.setTempoFactor(1.5f);

```

Sequencerをしたら、それをじてしてください

```
sequencer.close();
```


ベアメタルサウンド

javaでするときには、ほとんどベアメタルにすることもできます。このコードはのバイナリデータをOSオーディオバッファにきんでサウンドをします。このようなをみすためのとなをすることはにです。はにであるため、はほぼリアルタイムでするがあります。

そのため、このは、よりなサウンドサンプリングにはできません。なツールをするそのようには、よりいアプローチです。

のは、のにののをしてする。

```
public void rectangleWave(byte volume, int hertz, int msecs) {
    final SourceDataLine dataLine;
    // 24 kHz x 8bit, single-channel, signed little endian AudioFormat
    AudioFormat af = new AudioFormat(24_000, 8, 1, true, false);
    try {
        dataLine = AudioSystem.getSourceDataLine(af);
        dataLine.open(af, 10_000); // audio buffer size: 10k samples
    } catch (LineUnavailableException e) {
        throw new RuntimeException(e);
    }

    int waveHalf = 24_000 / hertz; // samples for half a period
    byte[] buffer = new byte[waveHalf * 20];
    int samples = msecs * (24_000 / 1000); // 24k (samples / sec) / 1000 (ms/sec) * time(ms)

    dataLine.start(); // starts playback
    int sign = 1;

    for (int i = 0; i < samples; i += buffer.length) {
        for (int j = 0; j < 20; j++) { // generate 10 waves into buffer
            sign *= -1;
            // fill from the jth wave-half to the j+1th wave-half with volume
            Arrays.fill(buffer, waveHalf * j, waveHalf * (j+1), (byte) (volume * sign));
        }
        dataLine.write(buffer, 0, buffer.length); //
    }
    dataLine.drain(); // forces buffer drain to hardware
    dataLine.stop(); // ends playback
}
```

なるをさせるよりされたのために、のがであり、によってはよりきなサンプルサイズがである。この、かなりなコードとなり、ここではされます。

なオーディオ

ハローオーディオローカルまたはインターネットのストレージからサウンドファイルをするJavaのののようにえます。されていない.wavファイルでもし、mp3ファイルやファイルのにはしないでください。

```
import java.io.*;
import java.net.URL;
import javax.sound.sampled.*;
```

```

public class SoundClipTest {

    // Constructor
    public SoundClipTest() {
        try {
            // Open an audio input stream.
            File soundFile = new File("/usr/share/sounds/alsa/Front_Center.wav"); //you could
also get the sound file with an URL
            AudioInputStream audioIn = AudioSystem.getAudioInputStream(soundFile);
            AudioFormat format = audioIn.getFormat();
            // Get a sound clip resource.
            DataLine.Info info = new DataLine.Info(Clip.class, format);
            Clip clip = (Clip)AudioSystem.getLine(info);
            // Open audio clip and load samples from the audio input stream.
            clip.open(audioIn);
            clip.start();
        } catch (UnsupportedAudioFileException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (LineUnavailableException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        new SoundClipTest();
    }
}

```

オンラインでオーディオをむ <https://riptutorial.com/ja/java/topic/160/オーディオ>

76: オートボクシング

き

Autoboxingは、Javaコンパイラがプリミティブとそれにするオブジェクトラッパークラスのものです。、int -> Integer、double -> Double ...のがのは、これをunboxingとびます。、これは、コレクションのオブジェクトをできないコレクションでされます。ボクシングプリミティブはコレクションでするにです。

オートボクシングでは、コードでにするとパフォーマンスのがするがあります。

- <http://docs.oracle.com/javase/1.5.0/docs/guide/language/autoboxing.html>
- のアンボクシングとボクシングはパフォーマンスのをもたらしめますか

Examples

intとIntegerをじでする

ユーティリティクラスでジェネリックをする、オブジェクトとしてされている、それらのプリミティブとしくないので、はそれほどにちません。

```
List<Integer> ints = new ArrayList<Integer>();
```

Java SE 7

```
List<Integer> ints = new ArrayList<>();
```

いなことに、してするintのわりにすることができるIntegerがなとき。

```
for (int i = 0; i < 10; i++)  
    ints.add(i);
```

ints.add(i);ステートメントはのものとです。

```
ints.add(Integer.valueOf(i));
```

また、Integer#valueOfプロパティは、キャッシュのにあるときにJVMによってキャッシュされるIntegerオブジェクトをするなど、されます。

これはのにもされます

- byteとByte
- shortとShort
- floatとFloat

- double と Double
- long と Long
- char と Character
- boolean と Boolean

しかし、あいまいなだけではがです。のコードをえてみましょう

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
ints.add(2);
ints.add(3);
ints.remove(1); // ints is now [1, 3]
```

java.util.List インタフェースには、remove(int index) List インタフェースメソッドと remove(Object o) java.util.Collection からされたメソッドのがまれています。この、ボクシングはわれず、remove(int index) がびされます。

のををつをオートボクシングによりじるなJavaコードのののの-128と127

```
Integer a = 127;
Integer b = 127;
Integer c = 128;
Integer d = 128;
System.out.println(a == b); // true
System.out.println(c <= d); // true
System.out.println(c >= d); // true
System.out.println(c == d); // false
```

これがこる>=は、にびしintValue() りintながら==ではなく、int。

デフォルトでは、Javaのは[-128, 127]でキャッシュされるため、==するのは、こののIntegersは、がじにじオブジェクトをするためです。キャッシングのは、-XX:AutoBoxCacheMax JVMオプションをしてできます。つまり、-XX:AutoBoxCacheMax=1000でプログラムをすると、のコードはtrue。

```
Integer a = 1000;
Integer b = 1000;
System.out.println(a == b); // true
```

ifのブールの

アンボックスのため、ifでBooleanをできます。

```
Boolean a = Boolean.TRUE;
if (a) { // a gets converted to boolean
    System.out.println("It works!");
}
```

それはのためにくwhile、do whileとにおけるforにもがまれます。

Booleanがnull、でNullPointerExceptionがスローされることにしてください。

アンボクシングによって**NullPointerException**がすることがある

このコードをコンパイルする

```
Integer arg = null;
int x = arg;
```

しかし、に2の`java.lang.NullPointerException`でクラッシュします。

は、プリミティブ`int`は`null`をつことができないということです。

これはのですが、にはよりされたでよくれます。 `NullPointerException`はにではなく、しばしばそのようなバグのにつことはほとんどありません。

`autoboxing`と`auto-unboxing`にして、にボックスされていないに`null`がないことをしてください。

オートボクシングのメモリとのオーバーヘッド

オートボクシングはかなりのメモリオーバーヘッドになるがあります。例えば

```
Map<Integer, Integer> square = new HashMap<Integer, Integer>();
for(int i = 256; i < 1024; i++) {
    square.put(i, i * i); // Autoboxing of large integers
}
```

のメモリをします6kののデータで60kb。

さらに、ボックスされたは、メモリにのラウンドトリップをとし、CPUキャッシュのをさせます。のでは、アクセスされるメモリは、メモリのくなるにある5つのなる、`HashMap`オブジェクト、2。マップの`Entry[] table`オブジェクト、`Entry`オブジェクト、4。エントリ`key`オブジェクトボックスングプリミティブキー、5.エントリ`value`オブジェクトボックスングプリミティブ。

```
class Example {
    int primitive; // Stored directly in the class `Example`
    Integer boxed; // Reference to another memory location
}
```

`boxed`は`primitive`のみにアクセスする2つのメモリアccessをとする。

このマップからデータをすると、なコード

```
int sumOfSquares = 0;
for(int i = 256; i < 1024; i++) {
    sumOfSquares += square.get(i);
}
```

のものとです。

```
int sumOfSquares = 0;
for(int i = 256; i < 1024; i++) {
    sumOfSquares += square.get(Integer.valueOf(i)).intValue();
}
```

、のコードは、`Map#get(Integer)` ごとに `Integer` オブジェクトのとガベージコレクションをいます。はのをしてください。

このオーバーヘッドをらすために、いくつかのライブラリは、ボクシングをとしないプリミティブのされたコレクションをします。ボクシングのオーバーヘッドをけることにえて、これらのコレクションではエン트리あたり4のメモリがになります。Javaホットスポットは、ヒープではなくスタックのオブジェクトをすることでオートボクシングをできるがありますが、メモリのオーバーヘッドとそのメモリをすることはできません。

Java 8ストリームには、ボクシングをとしない `IntStream` などのプリミティブデータのされたインタフェースもあります。

なJavaランタイムは、`valueOf` ファクトリメソッドおよびオートボクシングによってされる `Integer` およびそののプリミティブラッパーオブジェクトのなキャッシュをします。`Integer` の、このキャッシュのデフォルトのは-128+127です。JVMのには、キャッシュサイズ/をするためのJVMコマンドラインオプションがあります。

なるケース **Integer** と **int** をなくできる

ケース1メソッドののわりにしています。

メソッドがとしてラッパークラスのオブジェクトをとする、はそれぞれのプリミティブのにやりとりすることができます。

```
int i;
Integer j;
void ex_method(Integer i)//Is a valid statement
void ex_method1(int j)//Is a valid statement
```

ケース2りをしているとき

メソッドがプリミティブをすとき、するラッパークラスのオブジェクトはりとしてじでされ、もです。

```
int i;
Integer j;
int ex_method()
{...
return j;}//Is a valid statement
Integer ex_method1()
{...
return i;}//Is a valid statement
}
```

ケース3を。

にしてをうときはいつでも、それぞれのラッパークラスのプリミティブとオブジェクトはじでできます。

```
int i=5;
Integer j=new Integer(7);
int k=i+j;//Is a valid statement
Integer m=i+j;//Is also a valid statement
```

とし ラッパークラスのオブジェクトにをまたはりてることをれないでください。

ラッパークラスオブジェクトとプリミティブをじでしているに、ラッパークラスオブジェクトにをしたりりてたりすることをれることはありません。にnullポインターがするがあります。

```
public class Test{
    Integer i;
    int j;
    public void met()
    {j=i;//Null pointer exception
    SOP(j);
    SOP(i);}
    public static void main(String[] args)
    {Test t=new Test();
    t.go();//Null pointer exception
    }
```

のでは、オブジェクトのはりてられていないため、されていないため、にプログラムはNULLポインタになります。のかららかなように、objectのはしてされず、りてられません。

オンラインでオートボクシングをむ <https://riptutorial.com/ja/java/topic/138/オートボクシング>

77: オブジェクトクラスのメソッドとコンストラクタ

き

このドキュメンテーション・ページは、Javaクラス・[コンストラクタ](#)にすると、しくされたクラスのスーパークラス・`Object` からにされる[オブジェクト・クラス・メソッド](#)にするをするためのページです。

- `public final native Class <> getClass`
- `public final native void notify`
- `public final native void notifyAll`
- `public final native void waitlong timeout`は`InterruptedException`をスローします。
- `public final void waitthrows InterruptedException`
- `public final void waitlong timeout、int nanos`は、`InterruptedException`をスローします。
- `public native int hashCode`
- `public boolean equalsObject obj`
- `public String toString`
- `protected native Object clone`は`CloneNotSupportedException`をスローします。
- `protected void finalizeThrowable`をスローします。

Examples

`toString` メソッド

`toString()` メソッドは、オブジェクトのコンテンツをしてオブジェクトの`String`をするためにされます。このメソッドは、クラスをするときにオーバーライドするがあります。 `toString()` は、オブジェクトが`"hello " + anObject`ようににされたときにに`"hello " + anObject`ます。

のをしてください。

```
public class User {
    private String firstName;
    private String lastName;

    public User(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Override
    public String toString() {
        return firstName + " " + lastName;
    }
}
```



```

public static void main(String[] args) {
    User user = new User("John", "Doe");
    System.out.println(user.toString()); // Prints "John Doe"
}
}

```

ここで、`Object` クラスの `toString()` は、オブジェクトをするにオブジェクトにするのあるデータをするために、`User` クラスでオーバーライドされてい `User`。

`println()` をすると、オブジェクトの `toString()` メソッドがにびされます。したがって、これらのステートメントはじことをいします。

```

System.out.println(user); // toString() is implicitly called on `user`
System.out.println(user.toString());

```

の `User` クラスで `toString()` がオーバーライドされていない、`System.out.println(user)` は `User@659e0bfd` またはクラスのながほとんどないの `String` をすことがあります `User@659e0bfd`。これは、`println()` が `User` クラスのやビジネスルールについてもらない `Java Object` クラスの `toString()` をするためです。クラスでこのをしたいは、にメソッドをオーバーライドしてください。

equals メソッド

TL; DR

`==` ののテストそれらがじオブジェクトであるかどうか

`.equals()` はのしいかどうかをテストします に "しい" かどうか。

`equals()` は、2つのオブジェクトがしいかどうかをするためにされるメソッドです。 `Object` クラスの `equals()` メソッドのデフォルトのは、のがじインスタンスをしているにのみ `true` します。したがって、それは `==` によるとじようにします。

```

public class Foo {
    int field1, field2;
    String field3;

    public Foo(int i, int j, String k) {
        field1 = i;
        field2 = j;
        field3 = k;
    }

    public static void main(String[] args) {
        Foo foo1 = new Foo(0, 0, "bar");
        Foo foo2 = new Foo(0, 0, "bar");

        System.out.println(foo1.equals(foo2)); // prints false
    }
}

```

foo1

と `foo2` は同じフィールドでされていても、メモリの2つの異なるオブジェクトをしています。したがって、デフォルトの `equals()` は `false` され `false`。

オブジェクトの `equals()` をかするには、`equals()` をオーバーライドする必要があります。

```
public class Foo {
    int field1, field2;
    String field3;

    public Foo(int i, int j, String k) {
        field1 = i;
        field2 = j;
        field3 = k;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }

        Foo f = (Foo) obj;
        return field1 == f.field1 &&
            field2 == f.field2 &&
            (field3 == null ? f.field3 == null : field3.equals(f.field3));
    }

    @Override
    public int hashCode() {
        int hash = 1;
        hash = 31 * hash + this.field1;
        hash = 31 * hash + this.field2;
        hash = 31 * hash + (field3 == null ? 0 : field3.hashCode());
        return hash;
    }

    public static void main(String[] args) {
        Foo foo1 = new Foo(0, 0, "bar");
        Foo foo2 = new Foo(0, 0, "bar");

        System.out.println(foo1.equals(foo2)); // prints true
    }
}
```

ここでオーバーライドされた `equals()` メソッドは、フィールドが同じにオブジェクトがしいとします。

`hashCode()` メソッドもきされています。このメソッドのでは、2つのオブジェクトがしい、そのハッシュも同じでなければならないことがされています。そのため、`hashCode()` と `equals()` にオーバーライドする必要があります。

`equals` メソッドののにしてください。 `Foo obj` ではなく `Object obj Foo obj` です。をメソッドにれた、それは `equals` メソッドのオーバーライドではありません。

のクラスをくとき `equals()` と `hashCode()` オーバーライドするときのロジックをするがあります。ほとんどのIDEはにこれをします。

`equals()` のは、コアJava APIのである `String` クラスにあります。むしろポインタをするよりも、`String` クラスは、コンテンツの `String`。

Java SE 7

Java 1.7では `java.util.Objects` クラスがされました。このクラスは、2つの `null` をするなメソッド `equals` をするため、`equals` メソッドのをするためにできます。

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null || getClass() != obj.getClass()) {
        return false;
    }

    Foo f = (Foo) obj;
    return field1 == f.field1 && field2 == f.field2 && Objects.equals(field3, f.field3);
}
```

クラスの

`equals` メソッドはどのオブジェクトにしてもできるので、メソッドがしばしば `null` チェックしたににうことの1つは、されるオブジェクトのクラスがのクラスとするかどうかをチェックすることです。

```
@Override
public boolean equals(Object obj) {
    //...check for null
    if (getClass() != obj.getClass()) {
        return false;
    }
    //...compare fields
}
```

これは、、クラスオブジェクトをすることによってのようにわれま。しかし、それはらかでないかもしれないいくつかのなケースではするがあります。たとえば、いくつかのフレームワークはクラスのプロキシをし、これらのプロキシはにはなるクラスです。JPAをしたをにします。

```
Foo detachedInstance = ...
Foo mergedInstance = entityManager.merge(detachedInstance);
if (mergedInstance.equals(detachedInstance)) {
    //Can never get here if equality is tested with getClass()
    //as mergedInstance is a proxy (subclass) of Foo
}
```

そのをするための1つのメカニズムは、`instanceof` をしてクラスをすることです

```
@Override
public final boolean equals(Object obj) {
    if (!(obj instanceof Foo)) {
        return false;
    }
    //...compare fields
}
```

しかし、`instanceof` をする `instanceof` はけなければならないとしがいくつかあり `instanceof`。 `Foo` のは、にのサブクラスをつことができ、それらのサブクラスは、されるがありますので `equals()` すると、ケースに `Foo` にしい `FooSubclass` が、 `FooSubclass` しくない `Foo`。

```
Foo foo = new Foo(7);
FooSubclass fooSubclass = new FooSubclass(7, false);
foo.equals(fooSubclass) //true
fooSubclass.equals(foo) //false
```

これは、とのにするため、 `equals()` メソッドのなです。として、 `instanceof` をするとき `equals()` のように `equals()` メソッドを `final` にすることをお `equals()` ます。これにより、サブクラスが `equals()` をオーバーライドしたり、なにしないように `equals()` ます。

hashCode メソッド

Javaクラスが `equals` メソッドをオーバーライドする `equals`、 `hashCode` メソッドもオーバーライドするがあります。 **メソッドのでされている** とおり

- Javaアプリケーションのにじオブジェクトにしてびされると、オブジェクトの `equals` でされるがされないう、 `hashCode` メソッドはにじをすがあります。このは、アプリケーションの1のからじアプリケーションののまでしているはありません。
- `equals(Object)` メソッドによって2つのオブジェクトがしい、2つのオブジェクトのそれぞれで `hashCode` メソッドをびすと、じのがされなければなりません。
- `equals(Object)` メソッドによって2つのオブジェクトがしくない、2つのオブジェクトのそれぞれで `hashCode` メソッドをびすと、々のがされるはありません。しかし、プログラマは、なオブジェクトにしてのをすると、ハッシュテーブルのパフォーマンスがするがあることにするがあります。

ハッシュコードは、 `HashMap`、 `HashTable`、 `HashSet` などのハッシュでされます。 `hashCode` のは、オブジェクトがされるバケットをします。これらのハッシュは、された `hashCode` がであれば、よりです。い `hashCode` のなは、 `hashCode` のがであることです。いえれば、のインスタンスがじバケットにされるはいです。

ハッシュコードをするアルゴリズムは、のようになります。

```
public class Foo {
    private int field1, field2;
    private String field3;
```

```

public Foo(int field1, int field2, String field3) {
    this.field1 = field1;
    this.field2 = field2;
    this.field3 = field3;
}

@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null || getClass() != obj.getClass()) {
        return false;
    }

    Foo f = (Foo) obj;
    return field1 == f.field1 &&
        field2 == f.field2 &&
        (field3 == null ? f.field3 == null : field3.equals(f.field3));
}

@Override
public int hashCode() {
    int hash = 1;
    hash = 31 * hash + field1;
    hash = 31 * hash + field2;
    hash = 31 * hash + (field3 == null ? 0 : field3.hashCode());
    return hash;
}
}

```

Arrays.hashCodeをショートカットとしてする

Java SE 1.2

Java 1.2では、ハッシュコードをするアルゴリズムをするわりに、フィールドをむObjectまたはプリミティブをすることで、`java.util.Arrays#hashCode`をしてすることができます。

```

@Override
public int hashCode() {
    return Arrays.hashCode(new Object[] {field1, field2, field3});
}

```

Java SE 7

Java 1.7では`java.util.Objects`クラスがされました。`java.util.Objects`クラスは、されるオブジェクトのについてハッシュコードをするなメソッド`hash(Object... objects)`します。このメソッドは、`java.util.Arrays#hashCode`のようにします。

```

@Override
public int hashCode() {
    return Objects.hash(field1, field2, field3);
}

```

このはで、カスタム`hashCode()`メソッドがびされるたびにガベージオブジェクトがされます。

- `Object[]`がされます。 `Objects.hash()`バージョンでは、は "varargs"メカニズムによってされます。
- いずれかのフィールドがプリミティブである、それらはボックスされなければならない、よりなオブジェクトがされるがあります。
- にをするがあります。
- によってりさなければなら`Arrays.hashCode`は`Objects.hash`。
- `Arrays.hashCode`または`Objects.hash`がおそらくしなければならない`Object.hashCode()`のびしはインラインできません。

ハッシュコードのキャッシュ

オブジェクトのハッシュコードのはになるがあるため、にされたときにオブジェクトにハッシュコードをキャッシュすることはです。え

```
public final class ImmutableArray {
    private int[] array;
    private volatile int hash = 0;

    public ImmutableArray(int[] initial) {
        array = initial.clone();
    }

    // Other methods

    @Override
    public boolean equals(Object obj) {
        // ...
    }

    @Override
    public int hashCode() {
        int h = hash;
        if (h == 0) {
            h = Arrays.hashCode(array);
            hash = h;
        }
        return h;
    }
}
```

このアプローチは、ハッシュコードをキャッシュするなフィールドのオーバーヘッドにしてハッシュコードをりしするコストをします。これがパフォーマンスのとしてられるかどうかは、のオブジェクトがハッシュされるルックアップやそののによってまります。

また、`ImmutableArray`のハッシュコードがゼロ 2^{32} で1であれば、キャッシュはであることにつくでしょう。

に、このアプローチは、ハッシュしているオブジェクトがな、しくするのがはるかにしくなります。しかし、ハッシュコードがされるときながあります。のをしてください。

wait メソッドと notify メソッド

wait() と notify() のスレッドのびしが-タンデムで wait() のスレッドのびしがオブジェクトまでに、そのスレッドがブロックされます notify() または notifyAll() がじオブジェクトに。

wait/ notify

```
package com.example.examples.object;

import java.util.concurrent.atomic.AtomicBoolean;

public class WaitAndNotify {

    public static void main(String[] args) throws InterruptedException {
        final Object obj = new Object();
        AtomicBoolean aHasFinishedWaiting = new AtomicBoolean(false);

        Thread threadA = new Thread("Thread A") {
            public void run() {
                System.out.println("A1: Could print before or after B1");
                System.out.println("A2: Thread A is about to start waiting...");
                try {
                    synchronized (obj) { // wait() must be in a synchronized block
                        // execution of thread A stops until obj.notify() is called
                        obj.wait();
                    }
                    System.out.println("A3: Thread A has finished waiting. "
                        + "Guaranteed to happen after B3");
                } catch (InterruptedException e) {
                    System.out.println("Thread A was interrupted while waiting");
                } finally {
                    aHasFinishedWaiting.set(true);
                }
            }
        };

        Thread threadB = new Thread("Thread B") {
            public void run() {
                System.out.println("B1: Could print before or after A1");

                System.out.println("B2: Thread B is about to wait for 10 seconds");
                for (int i = 0; i < 10; i++) {
                    try {
                        Thread.sleep(1000); // sleep for 1 second
                    } catch (InterruptedException e) {
                        System.err.println("Thread B was interrupted from waiting");
                    }
                }

                System.out.println("B3: Will ALWAYS print before A3 since "
                    + "A3 can only happen after obj.notify() is called.");

                while (!aHasFinishedWaiting.get()) {
                    synchronized (obj) {
                        // notify ONE thread which has called obj.wait()
                        obj.notify();
                    }
                }
            }
        };
    }
}
```

```

    };

    threadA.start();
    threadB.start();

    threadA.join();
    threadB.join();

    System.out.println("Finished!");
}
}

```

いくつかの

```

A1: Could print before or after B1
B1: Could print before or after A1
A2: Thread A is about to start waiting...
B2: Thread B is about to wait for 10 seconds
B3: Will ALWAYS print before A3 since A3 can only happen after obj.notify() is called.
A3: Thread A has finished waiting. Guaranteed to happen after B3
Finished!

```

```

B1: Could print before or after A1
B2: Thread B is about to wait for 10 seconds
A1: Could print before or after B1
A2: Thread A is about to start waiting...
B3: Will ALWAYS print before A3 since A3 can only happen after obj.notify() is called.
A3: Thread A has finished waiting. Guaranteed to happen after B3
Finished!

```

```

A1: Could print before or after B1
A2: Thread A is about to start waiting...
B1: Could print before or after A1
B2: Thread B is about to wait for 10 seconds
B3: Will ALWAYS print before A3 since A3 can only happen after obj.notify() is called.
A3: Thread A has finished waiting. Guaranteed to happen after B3
Finished!

```

getClass メソッド

getClass() メソッドをして、オブジェクトのランタイムクラスタイプをつけることができます。のをしてください。

```

public class User {

    private long userID;
    private String name;

    public User(long userID, String name) {
        this.userID = userID;
        this.name = name;
    }
}

public class SpecificUser extends User {
    private String specificUserID;
}

```



```

    public SpecificUser(String specificUserID, long userID, String name) {
        super(userID, name);
        this.specificUserID = specificUserID;
    }
}

public static void main(String[] args){
    User user = new User(879745, "John");
    SpecificUser specificUser = new SpecificUser("1AAAA", 877777, "Jim");
    User anotherSpecificUser = new SpecificUser("1BBBB", 812345, "Jenny");

    System.out.println(user.getClass()); //Prints "class User"
    System.out.println(specificUser.getClass()); //Prints "class SpecificUser"
    System.out.println(anotherSpecificUser.getClass()); //Prints "class SpecificUser"
}

```

`getClass()` メソッドはもなクラスをします。これは、`getClass()` がびされた `anotherSpecificUser` に、りが `User class SpecificUser` よりもツリーのがいため、`class SpecificUser` です。

すべきは、`getClass` メソッドはのようにされています。

```
public final native Class<?> getClass();
```

`getClass` びしによってされるのは `Class<? extends T>` ここで、`T` は `getClass` がびされるオブジェクトのです。

すなわち、のものがコンパイルされます

```
Class<? extends String> cls = "".getClass();
```

clone メソッド

`clone()` メソッドは、オブジェクトのコピーをしてすためにされます。このメソッドは、があり、コピーコンストラクタやそののコピーを `clone()` ためにするがあるため、けるべきです。

メソッドをびすすべてのクラスは、`Cloneable` インタフェースをするがあります。

`Cloneable` インタフェースは、びすオブジェクトクラスが `Cloneable` しているかどうかをチェックする `native clone()` メソッドのをするためにされるなるタグインタフェースです。びしがこのインタフェースをしていない、`CloneNotSupportedException` がスローされます。

`Object` クラスはこのインタフェースをしていないため、びしオブジェクトがクラス `Object` は `CloneNotSupportedException` がスローされます。

クローンがしいようにするには、クローンのオブジェクトとはしているがあります。したがって、オブジェクトがされるにオブジェクトをするがあります。つまり、されるオブジェクトのをするなオブジェクトのいずれかをコピーすることによって、に「ディープコピー」をするをします。これがしくされていない、されたオブジェクトはしていないし、クローンのオブジェクト

とじなオブジェクトへののをちます。これは、のがのにをほすため、のにつながります。

```
class Foo implements Cloneable {
    int w;
    String x;
    float[] y;
    Date z;

    public Foo clone() {
        try {
            Foo result = new Foo();
            // copy primitives by value
            result.w = this.w;
            // immutable objects like String can be copied by reference
            result.x = this.x;

            // The fields y and z refer to a mutable objects; clone them recursively.
            if (this.y != null) {
                result.y = this.y.clone();
            }
            if (this.z != null) {
                result.z = this.z.clone();
            }

            // Done, return the new object
            return result;

        } catch (CloneNotSupportedException e) {
            // in case any of the cloned mutable fields do not implement Cloneable
            throw new AssertionError(e);
        }
    }
}
```

finalize メソッド

これは、Objectクラスのされたメソッドです。このメソッドは、オブジェクトがメモリからされるにオブジェクトのまたはクリーンアップをするためにされます。

ドキュメントによると、このメソッドは、ガベージコレクションがオブジェクトへのがなくなったとしたときに、オブジェクトのガベージコレクタによってびされます。

しかし、オブジェクトがまだであるか、オブジェクトがになったときにガベージコレクタがされない、finalize()メソッドがびされるというはありません。それで、このにらないがいいです。

Javaのコアライブラリでは、FileInputStream.javaがいくつつかりFileInputStream.java。

```
protected void finalize() throws IOException {
    if ((fd != null) && (fd != FileDescriptor.in)) {
        /* if fd is shared, the references in FileDescriptor
         * will ensure that finalizer is only called when
         * safe to do so. All references using the fd have
         * become unreachable. We can call close()
         */
        close();
    }
}
```

```
}  
}
```

この、そのリソースがにじられていない、リソースをじるのチャンスです。

に、あらゆるのアプリケーションで `finalize()` メソッドをするのはいとえられ、けるべきです。

ファイナライザは、リソースをするためのものではありません ファイルのクローズなど。ガベージコレクタは、ヒープスペースでシステムのがなにびされます。ファイルハンドルがしているときや、そののでびされることにすることはできません。

ファイナライザのされたユースケースは、したについてのオブジェクトにするためにしようとしているオブジェクトにするものです。こののために、 `java.lang.ref.WeakReference<T>` クラスのよりいメカニズムがしてい `java.lang.ref.WeakReference<T>`。 `finalize()` メソッドをするがあるとわれるは、わりに `WeakReference` をしてじをできるかどうかをべるがあります。これでがしないは、デザインをよりくえすがあります。

さらにむには、Joshua Bloch「Effective Java」ブックの `finalize()` メソッドにするがあります。

オブジェクトコンストラクタ

Javaのすべてのコンストラクタは、 `Object` コンストラクタをびすがあります。これは、 `super()` のびしでわれます。これはコンストラクタののでなければなりません。このは、のをするにオブジェクトをにヒープにできるようにするためです。

コンストラクタで `super()` びしをしないと、コンパイラはそれをあなたのためにれます。

したがって、これらの3つのはすべてにです

`super()` コンストラクタへのびし

```
public class MyClass {  
  
    public MyClass() {  
        super();  
    }  
}
```

`super()` コンストラクタへののびしで

```
public class MyClass {  
  
    public MyClass() {  
        // empty  
    }  
}
```

のコンストラクタで

```
public class MyClass {  
  
}
```

コンストラクタチェーンはどうですか

のコンストラクタをコンストラクタののとしてびすことはです。スーパーコンストラクタへのびしとのコンストラクタへのびしはとものでなければならぬため、にです。

```
public class MyClass {  
  
    public MyClass(int size) {  
  
        doSomethingWith(size);  
  
    }  
  
    public MyClass(Collection<?> initialValues) {  
  
        this(initialValues.size());  
        addInitialValues(initialValues);  
  
    }  
}
```

しいMyClass(Arrays.asList("a", "b", "c"))をびすとList-Argumentをつ2のコンストラクタがびされ、のコンストラクタにされますこれはにsuper()、にaddInitialValues(int size)をリストの2のサイズでびします。これは、のコンストラクターがじをうがあるコードのをらすためにされます。

どのようにのコンストラクタをびすのですか

のでは、new MyClass("argument")またはnew MyClass("argument", 0)びすことができます。つまり、メソッドのオーバーロードとじように、したコンストラクタになパラメータでコンストラクタをびします。

Objectクラスのコンストラクタではどうなりますか

デフォルトののコンストラクタ super()びしをくをつサブクラスではもこりません。

ののコンストラクターはにできますが、そうでないは、のコンストラクターがにされていないり、コンパイラーはそれをれます。

どのようにオブジェクトは、オブジェクトのコンストラクタからされますか

オブジェクトののはJVMにされています。Javaのすべてのコンストラクタは、インスタンスのをする<init>というなメソッドとしてされます。この<init>メソッドはコンパイラによってされ、<init>はJavaのなではないため、ですることはできません。

JVMはこの<init>メソッドをどのようにびすのですか

JVMはinvokespecialをして<init>メソッドをびし、されていないクラスインスタンスにしてのみびすことができます。

は、JVMとJavaをしてください。

- メソッドJVM - [JVMS - 2.9](#)
- コンストラクタ - [JLS - 8.8](#)

オンラインでオブジェクトクラスのメソッドとコンストラクタをむ

<https://riptutorial.com/ja/java/topic/145/オブジェクトクラスのメソッドとコンストラクタ>

78: オブジェクトのクローニング

にオブジェクトのフィールドがのオブジェクトをしている、クローニングはいにくいことがあります。フィールドつまりのオブジェクトへののみをコピーするのではなく、**ディープコピー**をしたいがあります。

のは**クローンがれている**ため、`Cloneable`インターフェイスをして`clone`メソッドをオーバーライドするに2えるがあります。`clone`メソッドは、`Cloneable`インターフェイスではなく、`Object`クラスでされているため、`Cloneable`はパブリックな`clone`メソッドがないため、インターフェイスとしてしません。その、`clone`をするはかくされ、くされます。えは、きされますクラス`clone`々、オーバーライド、クラスのすべてにしている`clone`。らはそうするようにされていませんし、そうでなければあなたのコードはをスローするかもしれません。

をするためのもっといは、コピーコンストラクタまたはコピーファクトリをすることです。[Joshua Blochの「なJava Item 11にクローンをオーバーライドする」](#)をしてください。

Examples

コピーコンストラクタをした

オブジェクトをするなは、コピーコンストラクタをすることです。

```
public class Sheep {

    private String name;

    private int weight;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    // copy constructor
    // copies the fields of other into the new object
    public Sheep(Sheep other) {
        this.name = other.name;
        this.weight = other.weight;
    }

}

// create a sheep
Sheep sheep = new Sheep("Dolly", 20);
// clone the sheep
Sheep dolly = new Sheep(sheep); // dolly.name is "Dolly" and dolly.weight is 20
```

Cloneable インタフェースをすることによるクローニング

Cloneable インタフェースをしてオブジェクトをする

```
public class Sheep implements Cloneable {

    private String name;

    private int weight;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

}

// create a sheep
Sheep sheep = new Sheep("Dolly", 20);
// clone the sheep
Sheep dolly = (Sheep) sheep.clone(); // dolly.name is "Dolly" and dolly.weight is 20
```

いコピーをするクローン

オブジェクトのクローンのデフォルトのは、オブジェクトのフィールドのシャローコピーをすることです。その、のオブジェクトとされたオブジェクトのは、じオブジェクトへのをします。

このは、そのをしています。

```
import java.util.List;

public class Sheep implements Cloneable {

    private String name;

    private int weight;

    private List<Sheep> children;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    public List<Sheep> getChildren() {
        return children;
    }

    public void setChildren(List<Sheep> children) {
```

```

        this.children = children;
    }

}

import java.util.Arrays;
import java.util.List;

// create a sheep
Sheep sheep = new Sheep("Dolly", 20);

// create children
Sheep child1 = new Sheep("Child1", 4);
Sheep child2 = new Sheep("Child2", 5);

sheep.setChildren(Arrays.asList(child1, child2));

// clone the sheep
Sheep dolly = (Sheep) sheep.clone();
List<Sheep> sheepChildren = sheep.getChildren();
List<Sheep> dollysChildren = dolly.getChildren();
for (int i = 0; i < sheepChildren.size(); i++) {
    // prints true, both arrays contain the same objects
    System.out.println(sheepChildren.get(i) == dollysChildren.get(i));
}

```

ディープコピーをするクローン

ネストされたオブジェクトをコピーするには、このように**コピー**をすることができます。

```

import java.util.ArrayList;
import java.util.List;

public class Sheep implements Cloneable {

    private String name;

    private int weight;

    private List<Sheep> children;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        Sheep clone = (Sheep) super.clone();
        if (children != null) {
            // make a deep copy of the children
            List<Sheep> cloneChildren = new ArrayList<>(children.size());
            for (Sheep child : children) {
                cloneChildren.add((Sheep) child.clone());
            }
            clone.setChildren(cloneChildren);
        }
        return clone;
    }
}

```



```

    public List<Sheep> getChildren() {
        return children;
    }

    public void setChildren(List<Sheep> children) {
        this.children = children;
    }
}

import java.util.Arrays;
import java.util.List;

// create a sheep
Sheep sheep = new Sheep("Dolly", 20);

// create children
Sheep child1 = new Sheep("Child1", 4);
Sheep child2 = new Sheep("Child2", 5);

sheep.setChildren(Arrays.asList(child1, child2));

// clone the sheep
Sheep dolly = (Sheep) sheep.clone();
List<Sheep> sheepChildren = sheep.getChildren();
List<Sheep> dollysChildren = dolly.getChildren();
for (int i = 0; i < sheepChildren.size(); i++) {
    // prints false, both arrays contain copies of the objects inside
    System.out.println(sheepChildren.get(i) == dollysChildren.get(i));
}

```

コピーファクトリをしたクローニング

```

public class Sheep {

    private String name;

    private int weight;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    public static Sheep newInstance(Sheep other) {
        return new Sheep(other.name, other.weight)
    }

}

```

オンラインでオブジェクトのクローニングをむ <https://riptutorial.com/ja/java/topic/2830/オブジェクトのクローニング>

79: オブジェクト

これは "Null Pointer Exception" をするのにつはずです。オブジェクトがnullなので、そのうちの1つをしますが、プログラムコードはそのオブジェクトでかをするををしています。しかし、それはのトピックにする...

Examples

メソッドパラメータとしてのオブジェクト

このトピックでは、オブジェクトのについてします。Javaでプログラミングをめてしたをにしています。クラス、メインメソッド、オブジェクトインスタンス、オブジェクトの "on" メソッドのびし、メソッドへのパラメータのしなど、いくつかのとにれているはずです。

```
public class Person {  
  
    private String name;  
  
    public void setName(String name) { this.name = name; }  
  
    public String getName() { return name; }  
  
    public static void main(String [] arguments) {  
        Person person = new Person();  
        person.setName("Bob");  
  
        int i = 5;  
        setPersonName(person, i);  
  
        System.out.println(person.getName() + " " + i);  
    }  
  
    private static void setPersonName(Person person, int num) {  
        person.setName("Linda");  
        num = 99;  
    }  
}
```

Javaプログラミングをにするためには、このをあなたのののかにすることができます。そのは、Javaのみをするでなものです。

ごのように、`person` オブジェクトをインスタンスする `main` があり、そのオブジェクトの `name` フィールドを "Bob" にするメソッドをびします。に、のメソッドをびし、`person` を2つのパラメータの1つとしてします。もう1つのパラメータは5にされたです。

びされたメソッドは、されたオブジェクトの `name` を "Linda" にし、されたを99にしてからります。

では、がされるのですか

だから、なぜ`person`なされたが`main`にされるのですかにえられたはなぜですか

びしがわれると、`main`メソッドは`person` オブジェクトを`setPersonName`メソッドに`setPersonName`ます。`setAnotherName`がそのオブジェクトにしてうはそのオブジェクトのであり、メソッドがるときにそれらのはそのオブジェクトのです。

じことをもうつの`person`はオブジェクトヒープにされているは、があればをします。メソッドがそのオブジェクトにしてうは、そのオブジェクトにしてわれ、をうメソッドがまだアクティブであるかされたかによってをけません。メソッドがすると、オブジェクトにえられたはすべてそのオブジェクトにされます。

されたとこれをしてください。これは**プリミティブ** `intInteger`オブジェクトインスタンスではないなので、された"によって"、そのはメソッドにされ、されたのへのポインタではありません。メソッドは、しかし、それはメソッドびしがわれたときにされるにはしません。

Javaでは、すべてのプリミティブがしされます。オブジェクトはしされます。つまり、オブジェクトへのポインタが、それらをけるメソッドにパラメータとしてされます。

らかではないことは、びされたメソッドがしいオブジェクトをし、それをパラメータの1つとしてすことはであることをします。メソッドがメソッドびしによってまたはにされたオブジェクトをすのは、メソッドのりとしてです。にそれがどうやってうまくいかないのか、どうしたらうまくいくのかをてみましょう。

たちのさなにもうつのをしましょう

```
private static void getAnotherObjectNot(Person person) {
    person = new Person();
    person.setName("George");
}
```

`main`にり、`setAnotherName`のびしので、このメソッドともう1つの`println`びしをびしましょう

```
getAnotherObjectNot(person);
System.out.println(person.getName());
```

はプログラムがします

```
Linda 5
Linda
```

ジョージをとっていたオブジェクトにはがこったのですかさて、されたパラメータはLindaへのポインタでした。`getAnotherObjectNot`メソッドがしいオブジェクトをしたとき、それはLindaオブジェクトへのをGeorgeオブジェクトへのにきえました。Lindaオブジェクトはまだヒープにしていますが、`main`メソッドはまだそれにアクセスできますが、`getAnotherObjectNot`メソッドはそれへのがないため、そのはもできません。コードのは、しいオブジェクトをしてすをしていたようです

が、そうであればしませんでした。

それががやりたいことであれば、メソッドからしくされたオブジェクトをすがあります。

```
private static Person getAnotherObject() {
    Person person = new Person();
    person.setName("Mary");
    return person;
}
```

それからこれをのようになでください

```
Person mary;
mary = getAnotherObject();
System.out.println(mary.getName());
```

プログラムのはのようになります。

```
Linda 5
Linda
Mary
```

のがあるプログラムがここにあります

```
public class Person {
    private String name;

    public void setName(String name) { this.name = name; }
    public String getName() { return name; }

    public static void main(String [] arguments) {
        Person person = new Person();
        person.setName("Bob");

        int i = 5;
        setPersonName(person, i);
        System.out.println(person.getName() + " " + i);

        getAnotherObjectNot(person);
        System.out.println(person.getName());

        Person person;
        person = getAnotherObject();
        System.out.println(person.getName());
    }

    private static void setPersonName(Person person, int num) {
        person.setName("Linda");
        num = 99;
    }

    private static void getAnotherObjectNot(Person person) {
        person = new Person();
        person.setMyName("George");
    }
}
```

```
private static person getAnotherObject() {  
    Person person = new Person();  
    person.setMyName("Mary");  
    return person;  
}  
}
```

オンラインでオブジェクトをむ <https://riptutorial.com/ja/java/topic/5454/オブジェクト>

80: オプション

き

`Optional`は、`null`のをむとまないがあるコンテナオブジェクトです。がする、`isPresent()`は`true`をし、`get()`はをします。

まれているのにするさらなるは、のような、さ`orElse()`がしないは、デフォルトをし、および`ifPresent()`がする、コードのブロックをします。

- `Optional.empty()`の`Optional`インスタンスをします。
- `Optional.of(value)`された`null`をつ`Optional`をします。されたが`null`の、`NullPointerException`がスローされます。
- `Optional.ofNullable(value)` `null`ののあるされたをつ`Optional`をします。

Examples

`Optional`がの、デフォルトをします。

`NoSuchElementException`スローするがあるので、に`Optional.get()`しないでください。

`Optional.orElse(T)`および`Optional.orElseGet(Supplier<? extends T>)`メソッドは、`Optional`がのにデフォルトをするをします。

```
String value = "something";

return Optional.ofNullable(value).orElse("defaultValue");
// returns "something"

return Optional.ofNullable(value).orElseGet(() -> getDefaultValue());
// returns "something" (never calls the getDefaultValue() method)
```

```
String value = null;

return Optional.ofNullable(value).orElse("defaultValue");
// returns "defaultValue"

return Optional.ofNullable(value).orElseGet(() -> getDefaultValue());
// calls getDefaultValue() and returns its results
```

`orElse`と`orElseGet`のないは、オプションがのでもにされたがされているはのみがされることです。したがって、`orElse`はにのみし、あらゆるのにづいてをすることはできません。

「`Optional`の`map()`メソッドをして、な`null`チェックをわずに`null`なるのあるをします。

`map()`および`filter()`は、にのみされる`Stream`とはなり、ちにされます。

```
public <U> Optional<U> map(Function<? super T,? extends U> mapper)
```

コード

```
String value = null;

return Optional.ofNullable(value).map(String::toUpperCase).orElse("NONE");
// returns "NONE"
```

```
String value = "something";

return Optional.ofNullable(value).map(String::toUpperCase).orElse("NONE");
// returns "SOMETHING"
```

マッピングがnullをすときにOptional.mapがのオプションをすので、いくつかのmapをnullなのとしてできます。これは、Nullセーフチェーンともばれます。

のをえてみましょう。

```
String value = foo.getBar().getBaz().toString();
```

getBar、getBaz、およびtoStringいずれかが、NullPointerExceptionスローするがあります。

OptionalをしてtoString()からをするのはのとおりです。

```
String value = Optional.ofNullable(foo)
    .map(Foo::getBar)
    .map(Bar::getBaz)
    .map(Baz::toString)
    .orElse("");
```

いずれかのマッピングがnullをした、のをします。

はのですが、しなります。いずれのマッピングもnullをさなかったにのみ、のをします。

```
Optional.ofNullable(foo)
    .map(Foo::getBar)
    .map(Bar::getBaz)
    .map(Baz::toString)
    .ifPresent(System.out::println);
```

がないはをスローする

まれているをするには、OptionalのorElseThrow()メソッドをします。されていないは、をスローします。これは、のをするをいて、get()びしとです。このメソッドは、スローされるをさなければならぬサブライヤをけります。

のでは、このメソッドはにまれるをします。

```
Optional optional = Optional.of("something");

return optional.orElseThrow(IllegalArgumentException::new);
// returns "something" string
```

2のでは、がされていないため、このメソッドはをスローします。

```
Optional optional = Optional.empty();

return optional.orElseThrow(IllegalArgumentException::new);
// throws IllegalArgumentException
```

メッセージがなをスローするは、ラムダをすることもできます。

```
optional.orElseThrow(() -> new IllegalArgumentException("Illegal"));
```

フィルタ

`filter()` は、とするにのみをすることをすためにされます。

それを `if (!somePredicate(x)) { x = null; }`。

コード

```
String value = null;
Optional.ofNullable(value) // nothing
    .filter(x -> x.equals("cool string")) // this is never run since value is null
    .isPresent(); // false
```

```
String value = "cool string";
Optional.ofNullable(value) // something
    .filter(x -> x.equals("cool string")) // this is run and passes
    .isPresent(); // true
```

```
String value = "hot string";
Optional.ofNullable(value) // something
    .filter(x -> x.equals("cool string")) // this is run and fails
    .isPresent(); // false
```

プリミティブのオプションのコンテナの

`OptionalDouble`、`OptionalInt`と`OptionalLong`などの`Optional`が、にプリミティブをラップするようにされています。

```
OptionalInt presentInt = OptionalInt.of(value);
OptionalInt absentInt = OptionalInt.empty();
```

にはがあるので、`null`のなはありません。のコンテナはをってチェックできます。


```
presentInt.isPresent(); // Is true.
absentInt.isPresent(); // Is false.
```

に、をするためにがします。

```
// Prints the value since it is provided on creation.
presentInt.ifPresent(System.out::println);

// Gives the other value as the original Optional is empty.
int finalValue = absentInt.orElseGet(this::otherValue);

// Will throw a NoSuchElementException.
int nonexistentValue = absentInt.getAsInt();
```

がするのみコードをする

```
Optional<String> optionalWithValue = Optional.of("foo");
optionalWithValue.ifPresent(System.out::println); // Prints "foo".

Optional<String> emptyOptional = Optional.empty();
emptyOptional.ifPresent(System.out::println); // Does nothing.
```

サプライヤをしてしてデフォルトをする

の `orElse` メソッドは `Object` とするため、 `Supplier` をするオプション `orElseGet` メソッドがあるがにう
かもしれません。

してください

```
String value = "something";
return Optional.ofNullable(value)
    .orElse(getValueThatIsHardToCalculate()); // returns "something"
```

オプションがではないため、がされていなくても `getValueThatIsHardToCalculate()` びします。

このペナルティをするには、サプライヤーにしてください

```
String value = "something";
return Optional.ofNullable(value)
    .orElseGet(() -> getValueThatIsHardToCalculate()); // returns "something"
```

このように、 `getValueThatIsHardToCalculate()` は、 `Optional` がのみにのみびされます。

フラットマップ

`flatMap` は `map` てい `map` 。このいは、javadocによってのようにされます。

このメソッドは `map(Function)` とてい `map(Function)` が、されたマップパーは `result` がすでに
`Optional` で、びされた、 `flatMap` はの `Optional` ラップしません。

あなたがすメソッドびしつまり、Optionalして、Optional.flatMapネストされたOptionals。

たとえば、のクラスがあるとします。

```
public class Foo {
    Optional<Bar> getBar(){
        return Optional.of(new Bar());
    }
}

public class Bar {
}
```

Optional.mapをすると、ネストされたOptionalがされます。 ie Optional<Optional<Bar>>。

```
Optional<Optional<Bar>> nestedOptionalBar =
    Optional.of(new Foo())
        .map(Foo::getBar);
```

ただし、Optional.flatMapをすると、なOptionalがされます。すなわち、Optional<Bar>。

```
Optional<Bar> optionalBar =
    Optional.of(new Foo())
        .flatMap(Foo::getBar);
```

オンラインでオプションをむ <https://riptutorial.com/ja/java/topic/152/オプション>

81: オラクルのコード

き

Javaプログラミングの[オラクルのスタイルガイド](#)は、Oracleのがうであり、のJavaがうことをしています。ファイル、ファイル、インデント、コメント、、、ステートメント、、、プログラミングプラクティスなどをみ、コードがまれています。

- のは、Oracleのしい[スタイル・ガイド](#)ににっています。いえれば、らはこのページのによつてにされていません。
- のスタイルガイドは、[のスタイルガイド](#)とのコードのとがあるようにににされています。
- のスタイルガイドは、Brian GoetzJava Language ArchitectとMark ReinholdJava Platformのチーフアーキテクトので[ピアレビュー](#)されています。
- はです。らはコードのしいをするつもりですが、コードをしくフォーマットするのがあるかもしれません。これはなです。ガイドラインにして、コードをフォーマットするにはいくつかがあります。

Examples

パッケージ

- パッケージは、アンダースコアやそののをくすべてにするがあります。
- パッケージは、のWebアドレスのでまいます。これは、プロジェクト/プログラムにするパッケージのにうことができます。
- をしないでください。 `java.lang.annotation`ではなく `java.lang.annotations` をするAPIのにつてください。
- `com.yourcompany.widget.button`、 `com.yourcompany.core.api`

クラス、インタフェース、の

- クラスとは、、、でなければなりません。
- インタフェースは、... ableでわるまたはでなければなりません。
- のののとをさせてします [CamelCase](#)。
- `^[AZ][a-zA-Z0-9]*$` してください。
- がロングフォームよりくされていないは、をし、のをけてください。
- がいクラスのであるは、をとしてフォーマットします。
- `ArrayList`、 `BigInteger`、 `ArrayIndexOutOfBoundsException`、 `Iterable`。

メソッド

メソッドは、またはアクションののでなければなりません

- `^[az][a-zA-Z0-9]*$` するがあります。
- ののととをしてください。
- `toString`、`hashCode`

はののととがするがあります

- `^[az][a-zA-Z0-9]*$` する
- さらに
- `elements`、`currentIndex`

タイプ

するがないなケースでは、を1つします。

- `^[AZ][0-9]?$` する
- もしあるがのものよりもであれば、`K`と`V`のキーとはマップにあり、`R`はのりののように、そうでなければ`T`います。
- のがするようなケースでは、でかれたいをし、アンダースコア `_` をしてをります。
- `T`、`V`、`SRC_VERTEX`

またはによってがである `static final` フィールドは、とアンダースコア `_` でってをります。

- `^[AZ][A-Z0-9]*(_[A-Z0-9]+)*$` する
- `BUFFER_SIZE`、`MAX_LEVEL`

にするそののガイドライン

- スコープのメソッド、をす/すことはけてください。
- のさをスコープのサイズにさせます。たとえば、きなクラスのフィールドにはわかりやすいをし、のローカルにはいをします。
- `public static` メンバーにをけるときは、にインポートされるとわれるは、をにします。
- しい [ネーミングセクション](#) Javaスタイルガイド

Oracleの[Javaスタイルガイドライン](#)

Javaソースファイル

- すべてのはLF、ASCII10でわらなければならない、えはCRやCR + LFでわらなければならないません。
- にがっていないがあります。
- ソースファイルのは、パッケージのプライベートクラスのみをむファイルであっても、そのにまれるクラスのと.javaがじでなければなりません。これは、package-info.javaなど、クラスをまないファイルにはされません。
- LFとはにはスペースASCII32のみです。これは、のたとえば、やりテラルなどをエスケープであることにしてください。
- \'、\"、\\、\t、\b、\r、\f、および\n えは、するまれるべきで\047 またはUnicode えは\u0027 をエスケープ。
- テストのためにのルールにするがある、テストはなをプログラムでするがあります。

パッケージ

```
package com.example.my.package;
```

パッケージのは、のされるをえているかどうかにかかわらず、をりしてはいけません。

インポートステートメント

```
// First java/javax packages
import java.util.ArrayList;
import javax.tools.JavaCompiler;

// Then third party libraries
import com.fasterxml.jackson.annotation.JsonProperty;

// Then project imports
import com.example.my.package.ClassA;
import com.example.my.package.ClassB;

// Then static imports (in the same order as above)
import static java.util.stream.Collectors.toList;
```

- インポートをソートするがあります...
 - ...に/でなインポートをにします。
 - ...のでパッケージでに
 - Javaパッケージ
 - javaxパッケージ
 - パッケージえは、org.xml
 -

パッケージcom.sun

。...にパッケージとクラスで3

- インポートステートメントは、のされるをえているかどうかにかかわらず、のりしをしないでください。
- のはしてはならない。

ワイルドカードのインポート

- ワイルドカードのインポートはにはされません。
- のにするクラスをインポートするえば、のなる「ノード」クラスをつツリーにビジターをするなど、ワイルドカードのインポートをできます。
- いずれのも、ファイルごとにのワイルドカードをインポートするはありません。

クラス

クラスメンバーの

クラスメンバーはのようにするがあります。

1. フィールド、、プライベート
2. コンストラクタ
3. ファクトリメソッド
4. の、された、の

にアクセスまたはによるフィールドとメソッドのけはありません。

こののをにします。

```
class Example {  
  
    private int i;  
  
    Example(int i) {  
        this.i = i;  
    }  
  
    static Example getExample(int i) {  
        return new Example(i);  
    }  
  
    @Override  
    public String toString() {  
        return "An example [" + i + "];"  
    }  
  
}
```

クラスメンバーのグループ

- するフィールドはにグループするがあります。
- ネストされたは、ののにすることができます。そののは、フィールドのにすることができます。
- コンストラクタとオーバーロードされたメソッドは、によってグループされ、するアリティでけされるべきです。これは、これらののがコードでにれることをします。
- コンストラクターは、のメンバーをさずにグループするがあります。
- オーバーロードされたメソッドのバリエーションは、のメンバーをさずにグループするがあります。

```
class ExampleClass {
    // Access modifiers first (don't do for instance "static public")
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}

interface ExampleInterface {
    // Avoid 'public' and 'abstract' since they are implicit
    void sayHello();
}
```

- はのでするがあります

- アクセス `public / private / protected`
- `abstract`
- `static`
- `final`
- `transient`
- `volatile`
- `default`
- `synchronized`
- `native`
- `strictfp`

- はのうちにきされるべきではありません。たとえば、インタフェースメソッドは`public`でも`abstract`でもしてはならず、ネストした`enum`とインタフェースをしてはいけません。
- メソッドのパラメータとローカルは、みやすさをさせたり、ののをしたりしないり、`final`されるべきではありません。
- フィールドは、それらをなながないり、`final`にするがあります。

インデント

- インデントレベルは**4**つのスペースです。
- インデントにできるのはスペースだけです。タブはありません。
- はインデントしてはいけません。これは、ののがないことによってされます。
- `case`は4つのスペースでインデントするがあり、ケースのステートメントはの4つのスペース

でインデントするがあります。

```
switch (var) {
  case TWO:
    setChoice("two");
    break;
  case THREE:
    setChoice("three");
    break;
  default:
    throw new IllegalArgumentException();
}
```

をインデントするのガイドラインについては、「[リステートメント](#)」をしてください。

ラッピングステートメント

- ソースコードとコメントは、ごとに80をえてはならず、インデントをめて1に100をえることはめったにありません。

は、ケースバイケースです。になのは、な ""とみやすさです。をにするとみにくくなります。に、「なみ」を80にめると、みにくくなるがあります。ここでするは、がモニターのをにをするのではなく、これらのなをできるようにすることをしています。

- URLまたはexampleコマンドはラップするべきではありません。

```
// Ok even though it might exceed max line width when indented.
Error e = isTypeParam
    ? Errors.InvalidRepeatableAnnotationNotApplicable(targetContainerType, on)
    : Errors.InvalidRepeatableAnnotationNotApplicableInContext(targetContainerType));

// Wrapping preferable
String pretty = Stream.of(args)
    .map(Argument::prettyPrint)
    .collectors(joining(", "));

// Too strict interpretation of max line width. Readability suffers.
Error e = isTypeParam
    ? Errors.InvalidRepeatableAnnotationNotApplicable(
        targetContainerType, on)
    : Errors.InvalidRepeatableAnnotationNotApplicableInContext(
        targetContainerType);

// Should be wrapped even though it fits within the character limit
String pretty = Stream.of(args).map(Argument::prettyPrint).collectors(joining(", "));
```

- いレベルでのラッピングは、よりいレベルでのラッピングよりもされます。
- 1につき1つのステートメントがです。
- は、の4つのいずれかでインデントするがあります
 - 1ののインデントにして8つのなスペースがあります。
 -

- 2 ラップされたのにして8つのなスペース。
- 3 のにアライメントされるそれがであることがらかであるり
- 4 したののメソッドびしにえられました。

ラッピングメソッド

```
int someMethod(String aString,
               List<Integer> aList,
               Map<String, String> aMap,
               int anInt,
               long aLong,
               Set<Number> aSet,
               double aDouble) {
    ...
}

int someMethod(String aString, List<Integer> aList,
               Map<String, String> aMap, int anInt, long aLong,
               double aDouble, long aLong) {
    ...
}

int someMethod(String aString,
               List<Map<Integer, StringBuffer>> aListOfMaps,
               Map<String, String> aMap)
    throws IllegalArgumentException {
    ...
}

int someMethod(String aString, List<Integer> aList,
               Map<String, String> aMap, int anInt)
    throws IllegalArgumentException {
    ...
}
```

- メソッドのは、をにリストするか、しいと+8なスペースをすることでフォーマットできます
- throwsをラップするがあるは、をthrowsのにき、のにして+8をインデントするか、またはのにして+8のどちらかでリストからつようにします。

のりし

- がにづくは、をりすのではなく、にの/にすることをしてください。
- のにれます。
- のにする。されたメソッドびし。

```
popupMsg("Inbox notification: You have "
        + newMsgs + " new messages");

// Don't! Looks like two arguments
popupMsg("Inbox notification: You have " +
        newMsgs + " new messages");
```

- 1つのをしります。
 - パッケージ
 - クラス
 - コンストラクタ
 - メソッド
 - イニシャライザ
 - インスタンスイニシャライザ
- ...のグループをするためにすることができます
 -
 - フィールド
 - ステートメント
- のするは、するメンバーのグループをするためにのみし、のメンバーのとしてはしないでください。

-
- のスペースをするがあります...
 - するとカッコからキーワードをするには
 - `lambda`のや`for`ループの`コロンラベル`の`コロン`のではないのようなシンボルのようなバインダリとのに、
 - `//`にコメントをします。
 - のりをカンマでって、`for`ループのをるセミコロン。
 - キャストのじの。
 - のでは、とをさせることはされません。
 - ごとに1つのごとにでも1つの
 - のは、`String args[]`ではなく、`String[] args`でなければなりません。
 - ローカルをにするにし、できるだけのくにします。

アノテーションは、アノテーションされているとはのにくがあります。

```
@SuppressWarnings("unchecked")
public T[] toArray(T[] typeHolder) {
    ...
}
```

ただし、のメソッドにをけるアノテーションは、をさせるは、メソッドとじにすることはできません。たとえば、のようにくことができます。

```
@Nullable String getName() { return name; }
```

とのから、すべてのをじにくか、を々のにくがあります。

```
// Bad.
@Deprecated @SafeVarargs
@CustomAnnotation
public final Tuple<T> extend(T... elements) {
    ...
}

// Even worse.
@Deprecated @SafeVarargs
@CustomAnnotation public final Tuple<T> extend(T... elements) {
    ...
}

// Good.
@Deprecated
@SafeVarargs
@CustomAnnotation
public final Tuple<T> extend(T... elements) {
    ...
}

// Good.
@Deprecated @SafeVarargs @CustomAnnotation
public final Tuple<T> extend(T... elements) {
    ...
}
```

ラムダ

```
Runnable r = () -> System.out.println("Hello World");

Supplier<String> c = () -> "Hello World";

// Collection::contains is a simple unary method and its behavior is
// clear from the context. A method reference is preferred here.
appendFilter(goodStrings::contains);

// A lambda expression is easier to understand than just tempMap::put in this case
trackTemperature((time, temp) -> tempMap.put(time, temp));
```

- ラムダは、のブロックラムダよりもましい。
- メソッドはラムダよりもにまれるべきです。
- バインドされたインスタンスメソッド、または1よりきいarityをつメソッドの、ラムダはしやすく、したがってましいものです。に、メソッドのがかららかでない。
- パラメータのは、みやすさをさせないりしてください。
- ラムダがにまたがるは、メソッドのをしてください。

なかっこ

```
return flag ? "yes" : "no";

String cmp = (flag1 != flag2) ? "not equal" : "equal";
```

```
// Don't do this
return (flag ? "yes" : "no");
```

- みやすさをさせるは、なグループにをえないをできます。
- グループは、なをむいではしなければならないが、よりいやなしでとがなをむにまれるべきである。なをうは、にする。
- `return` キーワードにくは、かっこでまれていてはいけません。

リテラル

```
long l = 5432L;
int i = 0x123 + 0xABC;
byte b = 0b1010;
float f1 = 1 / 5432f;
float f2 = 0.123e4f;
double d1 = 1 / 5432d; // or 1 / 5432.0
double d2 = 0x1.3p2;
```

- `long` リテラルでは、の_Lをするがあります。
- `16` リテラルでは、_{A F}をするがあります。
- のすべての、およびは、をするがあります。

ブレース

```
class Example {
    void method(boolean error) {
        if (error) {
            Log.error("Error occurred!");
            System.out.println("Error!");
        } else { // Use braces since the other block uses braces.
            System.out.println("No error");
        }
    }
}
```

- カッティングはカッコのではなく、カレントのにくがあります。
- ブロックがでないり、じのにしいがあるはずですのでいをしてください
- は、`if`や`loop`ののように、がオプションのでもされます。
 - ブロックがのコメントをむにまたがっているは、がです。
 - `if / else` ステートメントのブロックの1つにカッコがあるは、もうのブロックもです。
 - ブロックがむブロックのには、がです。
- `do...while` ループの `else`、`catch` および `while` キーワードは、のブロックのじとじになります。

ショートフォーム

```
enum Response { YES, NO, MAYBE }  
public boolean isReference() { return true; }
```

のは、をさせるためのものです/をさせるため。によっては、のガイドラインからした「い」もみやすく、わりにすることができます。これらのケースには、なenumやなメソッドやラムダなどがあります。

オンラインでオラクルのコードをむ <https://riptutorial.com/ja/java/topic/2697>オラクルのコード

82: カプセル

き

かなりなをつクラスがあり、コードからのプログラマがしたができないになったとします。これらのコードはコードにエラーをもたらしました。として、OOPでは、にされているオブジェクトのをメソッドをじてのみすることができます。オブジェクトのをし、オブジェクトメソッドをじてすべてのをすることは、データカプセルとしてられています。

を`private`マーキングし、にじて`public`することは、すでに`public`をすことよりもはるかにです。

カプセルがでないかもしれない1つのがあります "ダム"データをすることをのとするクラス

```
public class DumbData {  
    public String name;  
    public int timeStamp;  
    public int value;  
}
```

このは、クラスのインターフェイスは、それがするデータです。

`final`とマークされたは、にできないため、カプセルにすることなく`public`とマークすることができます。

Examples

をするためのカプセル

クラスには、インターフェイスとの2つのがあります。

インタフェースは、クラスのされたです。そのパブリックメソッドとは、インターフェイスのです。

は、クラスのです。のクラスは、クラスのについてるはありません。

カプセルとは、そのクラスののユーザーからのクラスのをすをします。これにより、クラスはにいてのをうことができます。

たとえば、`Angle`をすこのクラスをします。

```
public class Angle {  
  
    private double angleInDegrees;  
    private double angleInRadians;  
  
    public static Angle angleFromDegrees(double degrees){
```

```

        Angle a = new Angle();
        a.angleInDegrees = degrees;
        a.angleInRadians = Math.PI*degrees/180;
        return a;
    }

    public static Angle angleFromRadians(double radians){
        Angle a = new Angle();
        a.angleInRadians = radians;
        a.angleInDegrees = radians*180/Math.PI;
        return a;
    }

    public double getDegrees(){
        return angleInDegrees;
    }

    public double getRadians(){
        return angleInRadians;
    }

    public void setDegrees(double degrees){
        this.angleInDegrees = degrees;
        this.angleInRadians = Math.PI*degrees/180;
    }

    public void setRadians(double radians){
        this.angleInRadians = radians;
        this.angleInDegrees = radians*180/Math.PI;
    }
    private Angle(){}
}

```

このクラスはなまたは にします **angleInDegrees**と**angleInRadians**はにしています。クラスのメンバーがされている、の2つのがしているというはありません。

カプリングをするカプセル

カプセルをすると、クラスをびすコードにをえることなく、クラスをにすることができます。これにより、カプリングや、あるクラスがのクラスののにどれくらいしているかがします。

えは、Angleクラスののをのからしましょう

```

public class Angle {

    private double angleInDegrees;

    public static Angle angleFromDegrees(double degrees){
        Angle a = new Angle();
        a.angleInDegrees = degrees;
        return a;
    }

    public static Angle angleFromRadians(double radians){
        Angle a = new Angle();
        a.angleInDegrees = radians*180/Math.PI;
        return a;
    }
}

```

```

    }

    public double getDegrees(){
        return angleInDegrees;
    }

    public double getRadians(){
        return angleInDegrees*Math.PI / 180;
    }

    public void setDegrees(double degrees){
        this.angleInDegrees = degrees;
    }

    public void setRadians(double radians){
        this.angleInDegrees = radians*180/Math.PI;
    }

    private Angle(){}
}

```

このクラスのは、のを1つだけし、にじてのをするようにされました。

しかし、はされましたが、インタフェースはされませんでした。びしのクラスがangleInRadiansメソッドへのアクセスにしていたは、しいバージョンのAngleをするようにするがあります。クラスのびしは、クラスのをにすべきではありません。

オンラインでカプセルをむ <https://riptutorial.com/ja/java/topic/1295/カプセル>

83: カレンダーとそのサブクラス

Java 8、`Calendar`およびそのサブクラスは、`java.time`パッケージおよびそのサブパッケージによって与えられました。のAPIにカレンダーがなをき、それらをするがあります。

Examples

カレンダーオブジェクトの

`Calendar`オブジェクトは、`getInstance()`を`getInstance()`か、`GregorianCalendar`コンストラクターをしてできます。

これは、ケということにづくことができます。`Calendar` 1がされていることをし、ゼロをベースとしている`int`はに、よりいコードをするために0 `Calendar`のような、`Calendar.JANUARY`をけるために。

```
Calendar calendar = Calendar.getInstance();
Calendar gregorianCalendar = new GregorianCalendar();
Calendar gregorianCalendarAtSpecificDay = new GregorianCalendar(2016, Calendar.JANUARY, 1);
Calendar gregorianCalendarAtSpecificDayAndTime = new GregorianCalendar(2016, Calendar.JANUARY, 1, 6, 55, 10);
```

にのをしてください。のはをくがあります。たとえば、`Calendar.JANUARY`のは0

カレンダーフィールドの

`add()`および`roll()`をして`Calendar`フィールドをできます。

```
Calendar calendar = new GregorianCalendar(2016, Calendar.MARCH, 31); // 31 March 2016
```

`add()`メソッドはすべてのフィールドにをえ、カレンダーにのをまたはするにです

```
calendar.add(Calendar.MONTH, -6);
```

のによりカレンダーから6かがされ、2015930にります。

のフィールドにをえずにのフィールドをするには、`roll()`します。

```
calendar.roll(Calendar.MONTH, -6);
```

のは、のから6ケをし、そのは9としてされます。のフィールドはされていません。このではわりません。

AM / PMの

カレンダークラスをすると、AMまたはPMをにつけることができます。

```
Calendar cal = Calendar.getInstance();
cal.setTime(new Date());
if (cal.get(Calendar.AM_PM) == Calendar.PM)
    System.out.println("It is PM");
```

カレンダーの

2つのCalendarのいをするには、`getTimeInMillis()`メソッドをします。

```
Calendar c1 = Calendar.getInstance();
Calendar c2 = Calendar.getInstance();
c2.set(Calendar.DATE, c2.get(Calendar.DATE) + 1);

System.out.println(c2.getTimeInMillis() - c1.getTimeInMillis()); //outputs 86400000 (24 * 60 * 60 * 1000)
```

オンラインでカレンダーとそのサブクラスをむ <https://riptutorial.com/ja/java/topic/165/カレンダーとそのサブクラス>

84: クラス - Java Reflection

き

java.lang.Classクラスは、メタデータの、クラスのものの、にできるくのメソッドをします。

java.langパッケージとjava.lang.reflectパッケージは、Javaリフレクションのクラスをします。

それがされる

Reflection APIは、にのでされます。

IDE Eclipse、MyEclipse、NetBeansなどデバッガテストツールなど

Examples

ObjectクラスのgetClass メソッド

```
class Simple { }

class Test {
    void printName(Object obj){
        Class c = obj.getClass();
        System.out.println(c.getName());
    }
    public static void main(String args[]){
        Simple s = new Simple();

        Test t = new Test();
        t.printName(s);
    }
}
```

オンラインでクラス - Java Reflectionをむ <https://riptutorial.com/ja/java/topic/10151/クラス----java-reflection>

85: クラスとオブジェクト

き

オブジェクトにはとるいがあります。には、、、などのがあり、をったり、えたり、べたりしています。オブジェクトはクラスのインスタンスです。

クラス - クラスは、そのオブジェクトがサポートする/をするテンプレート/としてすることができます。

- `class Example {}` // クラスキーワード、

Examples

もなクラス

```
class TrivialClass {}
```

クラスはでも `class` キーワード、、およびからりちます。であってもかまいません。

`new` でクラスをインスタンスします。

```
TrivialClass tc = new TrivialClass();
```

オブジェクトメンバとメンバ

このクラスでは

```
class ObjectMemberVsStaticMember {  
  
    static int staticCounter = 0;  
    int memberCounter = 0;  
  
    void increment() {  
        staticCounter++;  
        memberCounter++;  
    }  
}
```

のコードスニペット

```
final ObjectMemberVsStaticMember o1 = new ObjectMemberVsStaticMember();  
final ObjectMemberVsStaticMember o2 = new ObjectMemberVsStaticMember();  
  
o1.increment();
```

```

o2.increment();
o2.increment();

System.out.println("o1 static counter " + o1.staticCounter);
System.out.println("o1 member counter " + o1.memberCounter);
System.out.println();

System.out.println("o2 static counter " + o2.staticCounter);
System.out.println("o2 member counter " + o2.memberCounter);
System.out.println();

System.out.println("ObjectMemberVsStaticMember.staticCounter = " +
ObjectMemberVsStaticMember.staticCounter);

// the following line does not compile. You need an object
// to access its members
//System.out.println("ObjectMemberVsStaticMember.staticCounter = " +
ObjectMemberVsStaticMember.memberCounter);

```

このをする

```

o1 static counter 3
o1 member counter 1

o2 static counter 3
o2 member counter 2

ObjectMemberVsStaticMember.staticCounter = 3

```

`static`メンバーはオブジェクトではなく、クラスでびすようにしてください。それはJVMのためのいはありませんが、のはそれをします。

`static`メンバーはクラスのであり、クラスごとに1のみします。インスタンスには`static`メンバーがし、インスタンスごとにしたコピーがあります。これは、メンバーにアクセスするためにそのクラスのオブジェクトにアクセスするがあることもします。

オーバーロードメソッド

々、なるのにしてじをするがあります。そのとき、じメソッドをなるパラメータセットですることができま。それぞれのなるパラメータセットはメソッドシグネチャとしてられています。にられるように、のメソッドはのをつことができます。

```

public class Displayer {

    public void displayName(String firstName) {
        System.out.println("Name is: " + firstName);
    }

    public void displayName(String firstName, String lastName) {
        System.out.println("Name is: " + firstName + " " + lastName);
    }

    public static void main(String[] args) {
        Displayer displayer = new Displayer();
    }
}

```

```

        displayer.displayName("Ram");           //prints "Name is: Ram"
        displayer.displayName("Jon", "Skeet"); //prints "Name is: Jon Skeet"
    }
}

```

は、2つのなるでじがびされることです。しているにってこのメソッドをびすこのは1つのか2つの、するメソッドがされます。

メソッドはオーバーロードすることができます

1. されたパラメータのについています。

method(String s) **および** method(String s1, String s2)。

2. パラメータのについています。

method(int i, float f) **および** method(float f, int i)。

りのをするだけでメソッドをオーバーロードすることはできません `int method()` は `String method()` と `String method()` なされ、されると `RuntimeException` がスローされます。りのをするは、オーバーロードするためにパラメータをするがあります。

なオブジェクトのと

オブジェクトはのクラスにっているので、なはですなはをしてください

```

public class Car {

    //Variables describing the characteristics of an individual car, varies per object
    private int milesPerGallon;
    private String name;
    private String color;
    public int numGallonsInTank;

    public Car(){
        milesPerGallon = 0;
        name = "";
        color = "";
        numGallonsInTank = 0;
    }

    //this is where an individual object is created
    public Car(int mpg, int, gallonsInTank, String carName, String carColor){
        milesPerGallon = mpg;
        name = carName;
        color = carColor;
        numGallonsInTank = gallonsInTank;
    }

    //methods to make the object more usable

    //Cars need to drive
    public void drive(int distanceInMiles){
        //get miles left in car
    }
}

```

```

        int miles = numGallonsInTank * milesPerGallon;

        //check that car has enough gas to drive distanceInMiles
        if (miles <= distanceInMiles){
            numGallonsInTank = numGallonsInTank - (distanceInMiles / milesPerGallon)
            System.out.println("Drove " + numGallonsInTank + " miles!");
        } else {
            System.out.println("Could not drive!");
        }
    }

    public void paintCar(String newColor){
        color = newColor;
    }

    //set new Miles Per Gallon
    public void setMPG(int newMPG){
        milesPerGallon = newMPG;
    }

    //set new number of Gallon In Tank
    public void setGallonsInTank(int numGallons){
        numGallonsInTank = numGallons;
    }

    public void nameCar(String newName){
        name = newName;
    }

    //Get the Car color
    public String getColor(){
        return color;
    }

    //Get the Car name
    public String getName(){
        return name;
    }

    //Get the number of Gallons
    public String getGallons(){
        return numGallonsInTank;
    }
}

```

オブジェクトはそのクラスのインスタンスです。したがって、オブジェクトをするは、メインクラスJavaではメインメソッド、AndroidではonCreate の2つののいずれかで Carクラスをびします。

オプション1

```

Car newCar = new Car(30, 10, "Ferrari", "Red");

```

オプション1は、に、オブジェクトのにCarにするすべてのことをプログラムにえます。のプロパティをするには、 repaintCar メソッドなどのメソッドの1つをびすがあります。

```

newCar.repaintCar("Blue");

```

メソッドにしいデータをしてください。のでは、データがしいり、repaintCarメソッドにをすこともできます。

これはオブジェクトのプロパティをするでした。オブジェクトのプロパティをけるには、り void ではないメソッドをしvoid をつCarクラスのメソッドをするがありvoid。

```
String myCarName = newCar.getName(); //returns string "Ferrari"
```

オプション1は、にすべてのオブジェクトのデータがあるになオプションです。

オプション2

```
`Car newCar = new Car();
```

オプション2はじをますが、オブジェクトをしにするにはさらにくのがです。はこのクラスをCarクラスでいしたいといいます

```
public void Car(){
    milesPerGallon = 0;
    name = "";
    color = "";
    numGallonsInTank = 0;
}
```

するオブジェクトににパラメータをすはありません。これは、オブジェクトのすべてのをとっていないが、っているをするがあるににです。これは、オブジェクトのインスタンスにデータをします。しないデータをびすと、エラーはスローされません。

していないオブジェクトのをでするがあることをれないでください。えば、

```
Car myCar = new Car();
String color = Car.getColor(); //returns empty string
```

これは、すべてのデータでされていないオブジェクトのでよくかけるいです。スタンドアロンのpublic Car(){ } でのCarオブジェクトをできるコンストラクタはしますが、myCarのはにはカスタマイズされていないため、エラーはされました。カーオブジェクトののしい

```
Car myCar = new Car();
myCar.nameCar("Ferrari");
myCar.paintCar("Purple");
myCar.setGallonsInTank(10);
myCar.setMPG(30);
```

また、メインクラスのメソッドをびしてオブジェクトのプロパティをすることをいしてください。

```
String myCarName = myCar.getName(); //returns string "Ferrari"
```


コンストラクタ

コンストラクターは、クラスのにがけられ、りのないメソッドであり、オブジェクトのにされます。メソッドのようなコンストラクターは、パラメーターをすることができます。コンストラクタはオブジェクトのにされます。クラスはコンストラクタもつことができます。

```
public class Hello{
    // constructor
    public Hello(String wordToPrint){
        printHello(wordToPrint);
    }
    public void printHello(String word){
        System.out.println(word);
    }
}
// instantiates the object during creating and prints out the content
// of wordToPrint
```

コンストラクタはいくつかのメソッドとはなることをすることができます。

1. コンストラクターは、`public`、`private`、および`protected`のみをすることができ、`abstract`、`final`、`static`、または`synchronized`することはできません。
2. コンストラクターにはりはありません。
3. コンストラクターはクラスとじをけなければなりませんMUST。 `Hello`では、`Hello`オブジェクトのコンストラクタはクラスとじです。
4. `this`キーワードはコンストラクタでのいがあります。 `this.method(...)`はのインスタンスのメソッドをびしますが、`this(...)`はのクラスのコンストラクタをしています。

コンストラクターは、`super`というキーワードをしてをしてびすこともできます。

```
public class SuperManClass{

    public SuperManClass(){
        // some implementation
    }

    // ... methods
}

public class BatmanClass extends SupermanClass{
    public BatmanClass(){
        super();
    }
    //... methods...
}
```

Java8および15.9をしてください。

staticをしてstatic finalフィールドをする

1つのをするがあるstatic finalフィールドをするには、staticをしてをりてます。のでは、なのStringします。

```
public class MyClass {

    public static final Set<String> WORDS;

    static {
        Set<String> set = new HashSet<>();
        set.add("Hello");
        set.add("World");
        set.add("foo");
        set.add("bar");
        set.add("42");
        WORDS = Collections.unmodifiableSet(set);
    }
}
```

メソッドのオーバーロードとオーバーライドの

メソッドのオーバーライドとオーバーロードは、Javaでサポートされている2のです。

メソッドのオーバーロード

メソッドのオーバーロードともばれますは、1つのクラスにじの2つまたはそのメソッドをつことができるです。はい、そのようにです。

```
public class Shape{
    //It could be a circle or rectangle or square
    private String type;

    //To calculate area of rectangle
    public Double area(Long length, Long breadth){
        return (Double) length * breadth;
    }

    //To calculate area of a circle
    public Double area(Long radius){
        return (Double) 3.14 * r * r;
    }
}
```

このようにして、ユーザーは、それがつのタイプにじて、じをエリアにしてびすことができます。

しかしのは、Javaコンパイラがどのメソッドをするのかをするです。

Javaでは、メソッド area() はじでものメソッドになるでもメソッドはじであることがわかりました。

オーバーロードされたメソッドは、なるリストとをたなければなりません。

それは、々はこのようなのをするのメソッドをすることはできないとわれています `public Double area(Long side)` この、のメソッドとし、Javaコンパイラのあいまいさをきこすからです。

、ありがとうございました。

りのがなるがあります。

アクセスがなるがあります。

なるをスローすることがあります。

なぜこれはとばれていますか

にはののとのコンパイルのについて、オーバーロードされるメソッドがびされるのはコンパイルにされるからです。

メソッドのオーバーロードをするなの1つは、コードがするシンプルさです。たとえば、ほとんどすべてののを `String.valueOf()` をえていますかシーンのろにかれているものはおそらくこのようなものです

```
static String valueOf(boolean b)
static String valueOf(char c)
static String valueOf(char[] data)
static String valueOf(char[] data, int offset, int count)
static String valueOf(double d)
static String valueOf(float f)
static String valueOf(int i)
static String valueOf(long l)
static String valueOf(Object obj)
```

メソッドオーバーライド

さて、メソッドのオーバーライドはい、それをしくする、それはなともばれますは、ややいなトピックです。

メソッドオーバーライドでは、クラスによってされたメソッドをきします。とったいいえをてみましょう。

```
public abstract class Shape{

    public abstract Double area(){
        return 0.0;
    }

}
```

そこでShapeというクラスがあり、それはおそらくのをすareaというメソッドをとっています。

CircleとRectangleという2つのクラスがあるとしましょう。

```

public class Circle extends Shape {
    private Double radius = 5.0;

    // See this annotation @Override, it is telling that this method is from parent
    // class Shape and is overridden here
    @Override
    public Double area(){
        return 3.14 * radius * radius;
    }
}

```

にクラス

```

public class Rectangle extends Shape {
    private Double length = 5.0;
    private Double breadth= 10.0;

    // See this annotation @Override, it is telling that this method is from parent
    // class Shape and is overridden here
    @Override
    public Double area(){
        return length * breadth;
    }
}

```

したがって、のクラスで `Shape` クラスによってされるメソッドがされました。はどのようにをることですかして、それは `psvm` をうことができます。

```

public class AreaFinder{

    public static void main(String[] args){

        //This will create an object of circle class
        Shape circle = new Circle();
        //This will create an object of Rectangle class
        Shape rectangle = new Rectangle();

        // Drumbeats .....
        //This should print 78.5
        System.out.println("Shape of circle : "+circle.area());

        //This should print 50.0
        System.out.println("Shape of rectangle: "+rectangle.area());

    }
}

```

うわーそれはらしいことではありませんかじの2つのオブジェクトがじメソッドをびし、なるをすの、それはのです。

これら2つのいをよりよくするためのチャートはのとおりです。

メソッドのオーバーロード	メソッドオーバーライド
メソッドのオーバーロードは、プログラムのをめるためにされます。	メソッドオーバーライドは、そのスーパークラスによってにされているメソッドののをするためにされます。
メソッドのオーバーロードはクラスでされます。	メソッドのオーバーライドは、IS-Aをつ2つのクラスでわれます。
メソッドのオーバーロードの、パラメータはなるがあります。	メソッドオーバーライドの、パラメータはじでなければなりません。
メソッドのオーバーロードは、コンパイルののです。	メソッドのオーバーライドはののです。
Javaでは、メソッドのりのをするだけでは、メソッドのオーバーロードをできません。りのは、メソッドのオーバーロードでじかなることができます。しかし、パラメータをするがあります。	りのは、メソッドのオーバーライドではじかでなければなりません。

オンラインでクラスとオブジェクトをむ <https://riptutorial.com/ja/java/topic/114/クラスとオブジェクト>

86: クラスパス

き

クラスパスは、Javaランタイムがクラスおよびリソースをすをリストします。クラスパスは、にコンパイルされたをつけるためにJavaコンパイラによってもされます。

Java クラスのロード

JVMJava Virtual Machineは、クラスがなときにクラスをロードしますこれはロードとばれます。されるクラスのは、の3つのでされます。

1. Java Class Libraryになものやそのなど、Java Platformでなものがにみられます。
2. クラスはにロードされますつまり、 `jre/lib/ext/`
3. クラスパスをしてユーザーのクラスがロードされます。

クラスは、 `java.lang.ClassLoader` サブタイプであるクラスをしてロードされ `java.lang.ClassLoader`。これについては、このトピック「[クラスローダー](#)」で詳しくしています。

クラスパス

クラスパスは、ユーザのクラスとパッケージのをするJVMまたはコンパイラによってされるパラメータです。これは、これらののほとんどのとにコマンドラインで、または `CLASSPATH` をしてできます。

Examples

クラスパスをするさまざまな

クラスパスをするは3つあります。

1. これは、 `CLASSPATH` をしてできます。

```
set CLASSPATH=...      # Windows and csh
export CLASSPATH=...    # Unix ksh/bash
```

2. コマンドラインでのようにすることができます

```
java -classpath ...
javac -classpath ...
```

`-classpath` または `-cp` オプションは、 `CLASSPATH` よりもされることにしてください。

3. なJARファイルのクラスパスは、 `MANIFEST.MF Class-Path` をしてします。

```
Class-Path: jar1-name jar2-name directory-name/jar3-name
```

これは、JARファイルがのようにされているにのみされることにしてください。

```
java -jar some.jar ...
```

このモードでは、JARファイルにClass-Pathがないでも、-classpathオプションとCLASSPATHはされます。

クラスパスがされていない、デフォルトのクラスパスはjava -jarをするはされたJARファイル、それのはのディレクトリです。

- <https://docs.oracle.com/javase/tutorial/deployment/jar/downman.html>
- <http://docs.oracle.com/javase/7/docs/technotes/tools/windows/classpath.html>

ディレクトリのすべての**JAR**をクラスパスにする

ディレクトリのすべてのJARをクラスパスにするは、クラスパスのワイルドカードをしてこれをにうことができます。えは

```
someFolder/*
```

これは、someFolderディレクトリのすべてのJARファイルとZIPファイルをクラスパスにするようにJVMにします。このは、なJARファイルのマニフェストファイルの-cp、CLASSPATH、またはClass-Pathでできます。クラスパスのクラスパスワイルドカードのと

ノート

1. Classpathワイルドカードは、Java 6でめてされました。のバージョンのJavaでは、「*」はワイルドカードとしてわれません。
2. ""ののにのをくことはできません。えは "someFolder / .jar"はワイルドカードではありません。
3. ワイルドカードは、 ".jar"または ".JAR"のファイルにのみします。 ZIPファイルはされ、のをつJARファイルもされます。
4. ワイルドカードは、ディレクトリのJARファイルにのみし、サブディレクトリにはしません。
5. JARファイルのグループがワイルドカードエントリとすると、クラスパスでのなはされません。

クラスパスのパス

クラスパスは、ディレクトリパス、JARまたはZIPファイルパス、またはJAR / ZIPワイルドカードであるのエントリです。

- コマンドラインでされたクラスパス -classpath またはとしては、のようにエントリをるがあ

ります; セミコロンは、Windowsの、または:のプラットフォームコロンそののLinux、UNIX、MacOSXのと。

- JARファイルのMANIFEST.MFのClass-Pathでは、のスペースをしてエントリをります。

には、クラスパスエントリにスペースをめむがある

- クラスパスがコマンドラインでされているは、なシェルをするだけです。えば

```
export CLASSPATH="/home/user/My JAR Files/foo.jar:second.jar"
```

は、するコマンドシェルによってなるがあります。

- クラスパスがJARファイルの "MANIFEST.MF"ファイルでされているは、URLエンコーディングをするがあります。

```
Class-Path: /home/user/My%20JAR%20Files/foo.jar second.jar
```

クラスパス

によっては、ネイティブコードがあり、JARのサブセットをするがあるなど、フォルダからすべてのJARをするだけではです。この、2つのmain() メソッドがです。のクラスローダーはクラスローダーをし、このクラスローダーをして2のmain() メソッドをびします。

ここでは、このプラットフォームのしいSWTネイティブJARをし、アプリケーションのJARをすべてしてから、のmain() メソッドをびします。[クロスプラットフォームJava SWTアプリケーションの](#)

クラスパスからリソースをロードする

JARにパッケージされているリソースイメージ、テキストファイル、プロパティ、KeyStoreなどをみむとです。こののために、ClassとClassLoader できます。

のプロジェクトがあるとします。

```
program.jar
|
\ -com
  \ -project
    |
    \ -file.txt
      \ -Test.class
```

また、Testクラスからfile.txtのにアクセスしたいとします。クラスローダーにねることで、そうすることができます

```
InputStream is = Test.class.getClassLoader().getResourceAsStream("com/project/file.txt");
```


クラスロードをすることにより、リソースパッケージのパスをすることができます。

あるいは、Testクラスオブジェクトにいわせることもできます

```
InputStream is = Test.class.getResourceAsStream("file.txt");
```

クラスオブジェクトをすると、パスはクラスにです。Test.classはcom.projectパッケージにあり、file.txtとじですが、パスをもするはありません。

しかし、のように、クラスオブジェクトからパスをすることはできます

```
is = Test.class.getResourceAsStream("/com/project/file.txt");
```

クラスをパスにマッピングする

のJavaツールチェーンおよびそれらとするようにされたサードパーティのツールは、クラスのをファイルのパスやそれをすのリソースにマッピングするためののルールをとっています。

マッピングはのとおりで

- デフォルトパッケージのクラスの、パスはなファイルです。
- きパッケージのクラスの、パッケージのコンポーネントはディレクトリにマップされます。
- ネストされたネストされたクラスとインナークラスの、クラスを\$ですることによってファイルコンポーネントがされます。
- のクラスでは、のわりにがされます。

これをのにします。

クラス	ソースパス	クラスファイルのパス
SomeClass	SomeClass.java	SomeClass.class
com.example.SomeClass	com/example/SomeClass.java	com/example/SomeClass.class
SomeClass.Inner	SomeClass.java	SomeClass\$Inner.class
クラスのSomeClass	SomeClass.java	SomeClass\$1.class、SomeClass\$2.classなど

クラスパスののみ

クラスパスのは、JVMにクラスやそののリソースをつけるをえることです。クラスパスとプロセスのはみっています。

クラスパスは、リソースをすためののをするパスのです。クラスパスでは、これらのは、ホストファイルシステムのディレクトリ、JARファイル、またはZIPファイルのいずれかです。いずれの

も、はされるのルートです。

クラスパスでクラスをするためのなはのとおりです。

1. クラスをクラスファイルパス_{RP} マップします。クラスのクラスファイルへのマッピングは、のでされています。
2. クラスパスのエントリ_Eについて
 - エントリがファイルシステムディレクトリの
 - _Eをにして_{RP}し、パス_{AP}します。
 - _{AP}がのファイルのパスであるかどうかをテストします。
 - はいのは、そのファイルからクラスをロードします
 - エントリがJARまたはZIPファイルの
 - JAR / ZIPファイルインデックスの_{RP}をしてください。
 - するJAR / ZIPファイル・エントリがするは、そのエントリからクラスをロードします。

クラスパスのリソースをするは、リソースパスがパスかパスかによってなります。リソースパスの、はのとおりです。 `Class.getResource`または`Class.getResourceAsStream`をしてされるなリソースパスの、クラスパッケージのパスはのにされます。

のJavaクラスローダーによってされたプロシージャーであることにしてください。カスタムクラスローダーはを々にすることがあります。

ブートストラップクラスパス

のJavaクラスローダーは、とアプリケーションクラスパスをチェックするに、まずブートストラップクラスパスでクラスをします。デフォルトでは、ブートストラップクラスパスは、"rt.jar"ファイルとJREインストールでされるそののなJARファイルでされています。これらは、のJava SEクラスライブラリのすべてのクラスと、さまざまな「」クラスをします。

のでは、これにするはありません。デフォルトでは、 `java`、 `javac`などのコマンドは、ランタイムライブラリのなバージョンをします。

にときどき、ライブラリのクラスのバージョンをして、Javaランタイムののをするがあります。たとえば、のでできないランタイムライブラリに「ショーSTOPパー」バグがするがあります。このようなでは、されたクラスをむJARファイルをし、それをJVMをするブートストラップクラスパスにすることができます。

`java` コマンドでは、ブートストラップクラスパスをするための`-X`オプションをしています。

- `-Xbootclasspath:<path>`は、のブートクラスパスをされたパスにきえます。
- `-Xbootclasspath/a:<path>`は、されたパスをのブートクラスパスにします。
- `-Xbootclasspath/p:<path>`は、されたパスをのブートクラスパスのに`-Xbootclasspath/p:<path>`ます。

Javaクラスそのをきえるまたはきするためにbootclasspathオプションをする、にJavaをしている
ことにしてください。コードをする、ライセンスのがあるかもしれません。 Java Binary License
のをし、にしてください。

オンラインでクラスパスをむ <https://riptutorial.com/ja/java/topic/3720/クラスパス>

87: クラスローダー

クラスローダは、アプリケーションがするクラスのとロードをすることとするクラスです。クラスローダーは、リソースをつけてロードすることもできます。

のクラスローダークラスは、クラスとリソースをファイルシステムのディレクトリとJARファイルとZIPファイルからロードできます。また、リモートサーバーからJARファイルとZIPファイルをダウンロードしてキャッシュすることもできます。

クラスローダーはチェーンされているため、JVMはアプリケーションのソースよりもクラス・ライブラリーのクラスをロードしようとします。カスタムクラスローダーをすると、プログラマーはこれをできます。また、バイトコードファイルのやバイトコードのなどもうことができます。

Examples

クラスローダーのインスタンスと

このなでは、アプリケーションがクラスローダをインスタンスしてクラスをにロードするをします。

```
URL[] urls = new URL[] {new URL("file:/home/me/extras.jar")};
ClassLoader loader = new URLClassLoader(urls);
Class<?> myObjectClass = loader.findClass("com.example.MyObject");
```

これでされたクラスローダーは、デフォルトのクラスローダーをとしてち、"extra.jar"をるにクラスローダーのクラスをつけようとします。されたクラスがにロードされている、`findClass`コールはにロードされたクラスへのをします。

`findClass`びしは、さまざまなでするがあります。もなのはのとおりです。

- されたクラスが見つからない、`ClassNotFoundException`をスローしてびし`ClassNotFoundException`。
- されたクラスが見つからないのクラスにする、そのびしは`NoClassDefFoundError`をスローします。

カスタムクラスローダーの

すべてのカスタムローダーは、`java.lang.ClassLoader`クラスをまたはにするがあり`java.lang.ClassLoader`。なポイントは、のメソッドです。

- `findClass(String)` - クラスローダーがクラスローディングのモデルにっているは、このメソッドをオーバーロードします。
- `loadClass(String, boolean)` - のモデルをするためにこのメソッドをオーバーロードします。
- `findResource`と`findResources` - これらのメソッドをオーバーロードして、リソースのみみを

カスタマイズします。

にバイトからクラスをロードするをう `defineClass` メソッドは、オーバーロードをぐために `final` ものです。 `defineClass` をびすにカスタムビヘイビアをするがあり `defineClass`。

バイトからのクラスをロードするなをにします。

```
public class ByteArrayClassLoader extends ClassLoader {
    private String classname;
    private byte[] classfile;

    public ByteArrayClassLoader(String classname, byte[] classfile) {
        this.classname = classname;
        this.classfile = classfile.clone();
    }

    @Override
    protected Class findClass(String classname) throws ClassNotFoundException {
        if (classname.equals(this.classname)) {
            return defineClass(classname, classfile, 0, classfile.length);
        } else {
            throw new ClassNotFoundException(classname);
        }
    }
}
```

`findClass` メソッドをオーバーライドしただけなので、このカスタムクラスローダーは `loadClass` がびされたときにのようにします。

1. クラスローダの `loadClass` メソッドは `findLoadedClass` をびして、こののクラスがこのクラスローダによってすでにロードされているかどうかをします。した、の `Class` オブジェクトはリクエストにされます。
2. に、 `loadClass` メソッドは、 `loadClass` びしをびすことによってクラスローダーにします。がをすることができる、オブジェクトは `Class` オブジェクトをし、にされます。
3. クラスローダーがクラスをロードできない、 `findClass` はオーバーライド `findClass` メソッドをびし、ロードされるクラスのをしします。
4. されたが `this.classname` とするは、 `defineClass` をびして `this.classfile` バイトからのクラスをロードします。の `Class` オブジェクトがされます。
5. がしない、 `ClassNotFoundException` をスローし `ClassNotFoundException`。

.class ファイルのロード

クラスをロードするには、にクラスをするがあります。クラスは `ClassLoader` によってされます。1つのがありますが、Oracleは `ClassLoader` のコードをこのをできるようにしていませんでした。クラスをするには、 `ClassLoader` プライベートメソッドである `defineClass()` というのメソッドにアクセスするがあります。

これにアクセスするには、しいクラス `ByteClassLoader` し、それを `ClassLoader` します。クラスを `ClassLoader` したので、 `ClassLoader` のプライベートメソッドにアクセスできます。 `defineClass()` できるようにするために、`private` の `defineClass()` メソッドのミラーのようにするしいメソッドをし

ます。たちは、クラス、になりますプライベートメソッドをびすには、`name`、クラスバイト、`classBytes`になりますのバイトのオフセットを、`0`ので`classBytes`のデータは、からまり
`classBytes[0]`そしてのバイトのオフセットたのだろう、`classBytes.length`はデータのサイズをす
ため、のオフセットになります。

```
public class ByteClassLoader extends ClassLoader {  
  
    public Class<?> defineClass(String name, byte[] classBytes) {  
        return defineClass(name, classBytes, 0, classBytes.length);  
    }  
  
}
```

さて、たちは`public defineClass()`メソッドをってい`defineClass()`。これは、クラスのとクラスの
バイトをとすこととびすことができます。

`stackoverflow`パッケージに`MyClass`というのクラスがあるとしましょう...

メソッドをびすにはクラスバイトがです。クラスの`Path`す`Path`オブジェクトを`Paths.get()`メソッド
をして`Paths.get()`、バイナリクラスのパスをとします。これで、クラスバイトを
`Files.readAllBytes(path)`でできます。そこで、`ByteClassLoader`インスタンスをし、したメソッド
`defineClass()`をします。たちはすでにクラスバイトをっていますが、メソッドをびすためにはク
ラスがです。このクラスはクラスこのは`stackoverflow.MyClass`のパッケージドットでえられます
。

```
Path path = Paths.get("MyClass.class");  
  
ByteClassLoader loader = new ByteClassLoader();  
loader.defineClass("stackoverflow.MyClass", Files.readAllBytes(path));
```

`defineClass()`メソッドは`Class<?>`オブジェクトをします。あなたがむなら、それをする事ができ
ます。

クラスをロードするには、`loadClass()`をびしてクラスをします。このメソッドは
`ClassNotFoundException`をスローして、`try catch`ブロックをするがあります

```
try{  
    loader.loadClass("stackoverflow.MyClass");  
} catch(ClassNotFoundException e){  
    e.printStackTrace();  
}
```

オンラインでクラスローダーをむ <https://riptutorial.com/ja/java/topic/5443/クラスローダー>

88: ゲッターとセッター

き

ここではゲッターとセッターについてします。Javaクラスのデータへのアクセスをするのです。

Examples

ゲッターとセッターの

カプセルは、OOPのです。これは、データとコードをのとしてラップすることにするものです。この、をprivateとしてし、GettersとSettersをしてアクセスしてをおよび/またはすることをおSettersします。

```
public class Sample {
    private String name;
    private int age;

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

これらのプライベートは、クラスのからアクセスすることはできません。したがって、それらはアクセスからされています。しかし、それらをまたはしたいは、ゲッターとセッターをできます。

getXxx() メソッドはxxxのをしますが、setXxx() をしてxxxをできます。

メソッドのはのとおりでvariableNameはvariableNameとばれvariableName。

- すべてのboolean

```
getVariableName() //Getter, The variable name should start with uppercase
setVariableName(..) //Setter, The variable name should start with uppercase
```

- boolean

```
isVariableName()      //Getter, The variable name should start with uppercase
setVariableName(...) //Setter, The variable name should start with uppercase
```

パブリックゲッターとセッターは、Java Beanの**プロパティ**のです。

setterまたはgetterをしてをする

セッターとゲッターは、オブジェクトがプライベートをむことをにします。プライベートは、きでアクセスおよびがです。えは、

```
public class Person {

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        if(name!=null && name.length()>2)
            this.name = name;
    }
}
```

このPersonクラスには、nameというのがあります。このにはgetName()メソッドをしてアクセスし、setName(String)メソッドをしてできますが、をするには、しいのさを2にするがあります。name publicにするのではなく、setterメソッドをすることで、のがのをつnameのをすることができます。ゲッターメソッドにもじことができます。

```
public String getName(){
    if(name.length()>16)
        return "Name is too large!";
    else
        return name;
}
```

のされたgetName()メソッドでは、さが16のにのみnameがされます。そうでないは、"Name is too large"がされます。これにより、プログラマは、クライアントクラスがましくないをするのをぎながら、でなをすることができます。

ゲッターとセッターをう

Javaでgetterとsetterをつオブジェクトをむクラスをえてみましょう

```
public class CountHolder {
    private int count = 0;

    public int getCount() { return count; }
    public void setCount(int c) { count = c; }
}
```


`count` はプライベートであるためアクセスできません。しかし、`getCount()` `setCount(int)` メソッドと `setCount(int)` メソッドはされているため、これらのメソッドにアクセスできます。いくつかの
にとって、これはをきこすかもしれません。なぜをするのですかなぜにそれらをするだけではないのですか

```
public class CountHolder {  
    public int count = 0;  
}
```

すべてのののために、これらの2つはまったくじでにじです。それらのいは、です。クラスのう
ことをえてみましょう

- まず、「には `int` をえるメソッドと、そののを `int` するメソッドがあります。
- に、「はあなたが `int` でしいものをにれることができます。」

これらはじようにこえるかもしれませんが、のものはにはその、もっとられています。それだけ
で、それは、あなたがそののとすることが出来ます。これはボールをにします。なやりとりが
どのようにこるかをしします。2のメソッドはからのをしています。のユーザーにはがかかりませ
んが、APIのはそのをすることをししますのAPIをリリースする。

カウントのとアクセスにアクセスをさせたいはしてください。は、これはです

```
public class CountHolder {  
    private int count = 0;  
  
    public synchronized int getCount() { return count; }  
    public synchronized void setCount(int c) { count = c; }  
}
```

2のでは、`count` がされているをししたりしすることなく、ほとんどになりました。これは、あ
なたがのによってされるように、ライブラリにしているがあるはさらにいことに、あなたはそ
のするがありませんし、のハードをすることをなくされています。

だからそれはをする。パブリックはこれまでずっといいことですなくとも、くないか

はです。では、のテストIE `System.out` さ `out` をてたパブリックのをすることができます。では、をす
ることは、オーバーヘッドをにえ、いをにさせることにはのももたらさない。ここでのガイド
ラインは、あなたがをすることをしているなら、のをってこれらののにしてそれをするがあります

1. このは、そのにおいてにするとえられるはないはずで。これはににねじれてしまうそして
それがいいといえども、がわるがあるので、`getters / setter` がなアプローチです。パブリッ
クをするは、にライブラリ/フレームワーク/APIでされているは、これをするがあります。
2. このはにされるがあり、をらすことによるのはそれをしします。は、メソッドをするためのオ
ーバーヘッドとをえるべきではないとえています。がにアプリケーションの99.9としている
ものについては、それほどできません。

おそらく、がのからしたとはっていないのことがあります。があるは、に `getters / setter` をしてく

ださい。

オンラインでゲッターとセッターをむ <https://riptutorial.com/ja/java/topic/3560/ゲッターとセッター>

89: コマンドラインの

- `public static void mainString [] args`

パラメーター

パラメーター	
args	コマンドライン。 <code>main</code> メソッドがJavaランチャーによって呼び出されると、 <code>args</code> は <code>null</code> でなく、 <code>null</code> もありません。

のJavaアプリケーションを `java` コマンドまたはのものをすると、 `args` のコマンドラインからして `main` メソッドが呼び出されます。

なことに、Java SEクラスライブラリは、コマンドのをサポートしていません。これにより、2つのがされます。

- Javaでをします。
- サードパーティのライブラリをする。

このトピックでは、なサードパーティのライブラリのをカバーしようとしています。のなリストについては、StackOverflowの [にするえ](#) をしてください。 ["Javaでコマンドラインをするは"](#)。

Examples

GWT ToolBaseによる

オプションのパラメータなど、よりなコマンドラインをするは、GoogleのGWTアプローチをするよりもです。すべてのクラスはされています

<https://riptide.com/>

`myprogram -dir "~/Documents" -port 8888` のコマンドライン、`myprogram -dir "~/Documents" -port 8888` をするはのとおりです。

```
public class MyProgramHandler extends ToolBase {
    protected File dir;
    protected int port;
    // getters for dir and port
    ...

    public MyProgramHandler() {
        this.registerHandler(new ArgHandlerDir() {
            @Override
```

```

        public void setDir(File dir) {
            this.dir = dir;
        }
    });
    this.registerHandler(new ArgHandlerInt() {
        @Override
        public String[] getTagArgs() {
            return new String[]{"port"};
        }
        @Override
        public void setInt(int value) {
            this.port = value;
        }
    });
}

public static void main(String[] args) {
    MyProgramHandler myShell = new MyProgramHandler();
    if (myShell.processArgs(args)) {
        // main program operation
        System.out.println(String.format("port: %d; dir: %s",
            myShell.getPort(), myShell.getDir()));
    }
    System.exit(1);
}
}

```

`ArgHandler`には、`isRequired()`メソッドがあります。このメソッドは、コマンドラインがであるとうためにきすることができますデフォルトのりは`false`で、はオプションです。

でをする

アプリケーションのコマンドラインがなは、コマンドコードのをすべてカスタムコードでうのがです。

このでは、のなケーススタディをします。いずれのも、がけられられない、コードはエラーメッセージをし、コマンドがしたことをシェルにえるために`System.exit(1)`をびします。いずれのも、Javaコードは、が`"myapp"`のラッパーをしてびされるとします。

のないコマンド

このケーススタディでは、コマンドはをとしません。このコードは、`args.length`がコマンドラインのをしていることをしています。

```

public class Main {
    public static void main(String[] args) {
        if (args.length > 0) {
            System.err.println("usage: myapp");
            System.exit(1);
        }
        // Run the application
        System.out.println("It worked");
    }
}

```

2つのをつコマンド

このケーススタディでは、コマンドはに2つのをとします。

```
public class Main {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println("usage: myapp <arg1> <arg2>");
            System.exit(1);
        }
        // Run the application
        System.out.println("It worked: " + args[0] + ", " + args[1]);
    }
}
```

`args.length` チェックをると、ユーザがコマンドラインをあまりにもなすぎてした、コマンドがクラッシュすることにしてください。

"フラグ"オプションとなくとも1つのをつコマンド

このケーススタディでは、コマンドには2つのオプションのフラグオプションがあり、オプションのになくとも1つのがです。

```
package tommy;
public class Main {
    public static void main(String[] args) {
        boolean feelMe = false;
        boolean seeMe = false;
        int index;
        loop: for (index = 0; index < args.length; index++) {
            String opt = args[index];
            switch (opt) {
                case "-c":
                    seeMe = true;
                    break;
                case "-f":
                    feelMe = true;
                    break;
                default:
                    if (!opts.isEmpty() && opts.charAt(0) == '-') {
                        error("Unknown option: '" + opt + "'");
                    }
                    break loop;
            }
        }
        if (index >= args.length) {
            error("Missing argument(s)");
        }

        // Run the application
        // ...
    }

    private static void error(String message) {
        if (message != null) {
```

```
        System.err.println(message);
    }
    System.err.println("usage: myapp [-f] [-c] [ <arg> ...]");
    System.exit(1);
}
}
```

てかるように、コマンドがなは、とオプションのがになります。 "コマンドライン"ライブラリを
することをおめします。 のをしてください。

オンラインでコマンドラインのをむ <https://riptutorial.com/ja/java/topic/4775/コマンドラインの>

90: コレクション

き

`java.util`のコレクションフレームワークは、のではできないをつのデータのクラスをします。

`Collections`フレームワークには、`Collection<O>`メインサブインタフェース`List<O>`と`Set<O>`、およびマッピングコレクション`Map<K, V>`インタフェースがまれています。コレクションはルートインタフェースであり、のくのコレクションフレームワークによってされています。

コレクションは、そののにオブジェクトのコレクションをできるオブジェクトです。ジェネリックスをして、コレクションにされているデータのをできます。

コレクションは、`java.util`または`java.util.concurrent`をします。

Java SE 1.4

Java 1.4.2はジェネリックスをサポートしていません。そのため、コレクションにまれるパラメータをすることはできません。のをたないことにえて、キャストをしてコレクションからしいをすがあります。

`Collection<E>`えて、のなコレクションがあり、そのうちのいくつかにはサブタイプがあります。

- `List<E>`は、オブジェクトのけられたコレクションです。これはにていますが、サイズのをしていません。は、しいにするためににサイズがきくなります。
- `Set<E>`はをしないオブジェクトのです。
 - `SortedSet<E>`はのけもする`Set<E>`です。
- `Map<K, V>`はキーとのペアのです。
 - `SortedMap<K, V>`は、のけもする`Map<K, V>`です。

Java SE 5

Java 5では、しいコレクションがされました。

- `Queue<E>`は、のでされるのです。これは、これがFIFOかLIFOかをします。これは、`Stack`クラスをします。

Java SE 6

Java 6では、いくつかのしいコレクションのサブタイプがされています。

- `NavigableSet<E>`は、なナビゲーションメソッドがみまれた`Set<E>`です。
- `NavigableMap<K, V>`は、なナビゲーションメソッドをみんだ`Map<K, V>`です。
- `Deque<E>`はどちらのからでもみれる`Queue<E>`です。

のはすべてインターフェイスです。それらをするには、 `ArrayList`、 `HashSet`、 `HashMap`、 `PriorityQueue` などのなクラスをつけるがあります。

タイプのコレクションには、パフォーマンスメトリックとユースケースがなるのがあります。

[Liskov Substitution Principle](#)がコレクションのサブタイプにされることにしてください。それは、され `SortedSet<E>` してにすことができます `Set<E>`。クラスのでコレクションをするのについては、Genericsセクションの[Bounded Parameters](#)をするとです。

のコレクションをするは、インタフェースをするわりにクラス `AbstractList` などのいずれかをするがです。

Java SE 1.2

1.2よりのバージョンでは、わりにのクラス/インタフェースをするがありました。

- `ArrayList` わりに `Vector`
- `Map` わりに `Dictionary`。 `Dictionary`はインタフェースではなくクラスでもあることにしてください。
- `HashMap` わりに `Hashtable`

これらのクラスはされており、のコードではしないでください。

Examples

`ArrayList`のとオブジェクトの

たちは、することができます `ArrayList` の `List` インタフェース

```
List aListOfFruits = new ArrayList();
```

Java SE 5

```
List<String> aListOfFruits = new ArrayList<String>();
```

Java SE 7

```
List<String> aListOfFruits = new ArrayList<>();
```

に、メソッド `add` をして `String` をします。

```
aListOfFruits.add("Melon");  
aListOfFruits.add("Strawberry");
```

のでは、 `ArrayList` はインデックス0に `String` "Melon"を、インデックス1に `String` "Strawberry"をみます。

`addAll(Collection<? extends E> c)` メソッドをしてのをもすることもできます

```
List<String> aListOfFruitsAndVeggies = new ArrayList<String>();
aListOfFruitsAndVeggies.add("Onion");
aListOfFruitsAndVeggies.addAll(aListOfFruits);
```

では "Onion" は `aListOfFruitsAndVeggies` 0 インデックスにされ、 "Melon" はインデックス1にあり、 "Strawberry" はインデックス2にあります。

のデータからコレクションをする

コレクション

Java Collections フレームワーク

々のデータから `List` をするなは、 `java.util.Arrays` メソッドをすること `Arrays.asList`

```
List<String> data = Arrays.asList("ab", "bc", "cd", "ab", "bc", "cd");
```

すべてのコレクションは、にしいコレクションにすべてのををするとしてのコレクションをるコンストラクタをします。

```
List<String> list = new ArrayList<>(data); // will add data as is
Set<String> set1 = new HashSet<>(data); // will add data keeping only unique values
SortedSet<String> set2 = new TreeSet<>(data); // will add data keeping unique values and sorting
Set<String> set3 = new LinkedHashSet<>(data); // will add data keeping only unique values and preserving the original order
```

Google Guava Collections フレームワーク

もうつのらしいフレームワークは、さまざまなタイプのコレクションをするためのらしいユーティリティクラスなメソッドをするである `Google Guava` です。 `Lists` と `Sets`

```
import com.google.common.collect.Lists;
import com.google.common.collect.Sets;
...
List<String> list1 = Lists.newArrayList("ab", "bc", "cd");
List<String> list2 = Lists.newArrayList(data);
Set<String> set4 = Sets.newHashSet(data);
SortedSet<String> set5 = Sets.newTreeSet("bc", "cd", "ab", "bc", "cd");
```

コレクションのマッピング

Java Collections フレームワーク

`Map<String, Object> map` とに、`Map<String, Object> map` すると、のようにすべてのをむしいマップをできます。

```
Map<String, Object> map1 = new HashMap<>(map);
SortedMap<String, Object> map2 = new TreeMap<>(map);
```

Apache Commons Collections フレームワーク

い Apache Commons あなたがをしてをすることができ `ArrayUtils.toMap` など `MapUtils.toMap`

```
import org.apache.commons.lang3.ArrayUtils;
...
// Taken from org.apache.commons.lang.ArrayUtils#toMap JavaDoc

// Create a Map mapping colors.
Map colorMap = MapUtils.toMap(new String[][] {{
    {"RED", "#FF0000"},
    {"GREEN", "#00FF00"},
    {"BLUE", "#0000FF"}}});
```

のは、なくとも2つのをむ `Map.Entry` または `Array` のいずれかでなければなりません。のはキーとしてされ、2のはとしてされます。

Google Guava Collections フレームワーク

Google Guava フレームワークのユーティリティクラスのは `Maps` です。

```
import com.google.common.collect.Maps;
...
void howToCreateMapsMethod(Function<? super K,V> valueFunction,
    Iterable<K> keys1,
    Set<K> keys2,
    SortedSet<K> keys3) {
    ImmutableMap<K, V> map1 = toMap(keys1, valueFunction); // Immutable copy
    Map<K, V> map2 = asMap(keys2, valueFunction); // Live Map view
    SortedMap<K, V> map3 = toMap(keys3, valueFunction); // Live Map view
}
```

Java SE 8

`Stream` をして、

```
Stream.of("xyz", "abc").collect(Collectors.toList());
```

または

```
Arrays.stream("xyz", "abc").collect(Collectors.toList());
```

リスト

ソースリストをせずにリストにするには、のことができます。

のアプローチ。よりくのががあるが、しやすい

```
List<String> newList = new ArrayList<String>();  
newList.addAll(listOne);  
newList.addAll(listTwo);
```

2のアプローチ。1はないがみやすい。

```
List<String> newList = new ArrayList<String>(listOne);  
newList.addAll(listTwo);
```

3のアプローチ。サードパーティの[Apache commons-collections](#)ライブラリがです。

```
ListUtils.union(listOne, listTwo);
```

Java SE 8

ストリームをすると、

```
List<String> newList = Stream.concat(listOne.stream(),  
listTwo.stream()).collect(Collectors.toList());
```

。 [インタフェースリスト](#)

ループのリストからをする

ループでをリストからするのはいいのですが、これはリストのインデックスとさがされたためです。

のリストでは、しないをもたらすとしいを与えるをします。

```
List<String> fruits = new ArrayList<String>();  
fruits.add("Apple");  
fruits.add("Banana");  
fruits.add("Strawberry");
```

for のりしでの **"Banana"** スキップ

コードサンプルはAppleとStrawberryのみをします。Bananaは、Appleがされるとインデックス₀にするのでスキップされますが、に_iは₁にやされます。

```
for (int i = 0; i < fruits.size(); i++) {
    System.out.println (fruits.get(i));
    if ("Apple".equals(fruits.get(i))) {
        fruits.remove(i);
    }
}
```

されたforステートメントでのをスローする

コレクションをりしてにするためです。

`java.util.ConcurrentModificationException`

```
for (String fruit : fruits) {
    System.out.println(fruit);
    if ("Apple".equals(fruit)) {
        fruits.remove(fruit);
    }
}
```

しい

Iteratorをしたwhileループでの

```
Iterator<String> fruitIterator = fruits.iterator();
while(fruitIterator.hasNext()) {
    String fruit = fruitIterator.next();
    System.out.println(fruit);
    if ("Apple".equals(fruit)) {
        fruitIterator.remove();
    }
}
```

Iteratorインターフェースには、このにのみみまれた`remove()`メソッドがあります。ただし、このメソッドはドキュメントでは「オプション」としてマークされており、

`UnsupportedOperationException`がスローされることがあります。

UnsupportedOperationException - このイテレータで`remove`がサポートされていない

したがって、このがサポートされていることをするためにドキュメントをチェックすることをおめしますには、コレクションがサードパーティのライブラリや`Collections.unmodifiable...()`メソッドのいずれかをしてされたのものでないり、はほぼにサポートされています。

Iteratorをしている、IteratorがされたときからListの`modCount`がされると、`ConcurrentModificationException`がスローされます。これは、じスレッドまたはじリストをするマルチスレッドアプリケーションでしているがあります。

`modCount` は、このリストがにされたをえる `int` です。のは、に、 `Collection` オブジェクトにして `add()` または `remove()` がびされることをします `Iterator` によるはカウントされません。 `Iterator` がされると、この `modCount` され、の `modCount` が `Iterator` がされたときとじであるかどうか、および `Iterator` がされたときに `List` すべてののりしにチェックされます。 `modCount` にかある、 `ConcurrentModificationException` スローされます。

したがって、のされたリストの、のようなのはをスローしません。

```
Iterator<String> fruitIterator = fruits.iterator();
fruits.set(0, "Watermelon");
while(fruitIterator.hasNext()){
    System.out.println(fruitIterator.next());
}
```

しかし、 `Iterator` をしたにしいを `List` にすると、 `ConcurrentModificationException` がスローされます。

```
Iterator<String> fruitIterator = fruits.iterator();
fruits.add("Watermelon");
while(fruitIterator.hasNext()){
    System.out.println(fruitIterator.next());    //ConcurrentModificationException here
}
```

```
for (int i = (fruits.size() - 1); i >=0; i--) {
    System.out.println (fruits.get(i));
    if ("Apple".equals(fruits.get(i))) {
        fruits.remove(i);
    }
}
```

これはもスキップしません。このアプローチのは、がであることです。ただし、ほとんどの、ではないアイテムをします。あなたは [LinkedList](#) これをうべきではありません。

、ループインデックスの

```
for (int i = 0; i < fruits.size(); i++) {
    System.out.println (fruits.get(i));
    if ("Apple".equals(fruits.get(i))) {
        fruits.remove(i);
        i--;
    }
}
```

これはもスキップしません。 `i` のが `List` からされると、にインデックス `i+1` するがしい `i` のになります。したがって、のがスキップすることなくのをするために、ループは `i` をすることができます。

「すべき」リストをする

```

ArrayList shouldBeRemoved = new ArrayList();
for (String str : currentArrayList) {
    if (condition) {
        shouldBeRemoved.add(str);
    }
}
currentArrayList.removeAll(shouldBeRemoved);

```

このソリューションをすると、ほしいがよりクリーンなでされているかどうかをできます。

Java SE 8

Java 8では、のがです。がループでわれるがない、これらはよりクリーンでよりです。

ストリームのフィルタリング

List をストリーミングしてフィルタリングすることができます。なフィルターをして、ましくな
いをすべてすることができます。

```

List<String> filteredList =
    fruits.stream().filter(p -> !"Apple".equals(p)).collect(Collectors.toList());

```

ここののすべてのとはなり、このではしい List インスタンスがされ、の List されません。

removeIf

ストリームのになのは、のアイテムをすることだけです。

```

fruits.removeIf(p -> "Apple".equals(p));

```

なコレクション

なのためにのあるコードのにつながるがあるため、コレクションをするのはいいではないがあります。「みり」コレクションをするために、Javaはそのなバージョンをします。

なコレクションは、コレクションができないことをするなコレクションのコピーであることがよくあります。しようとすると、UnsupportedOperationExceptionがします。

コレクションにするオブジェクトはききでることにすることができます。

```

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class MyPojoClass {
    private List<Integer> intList = new ArrayList<>();

    public void addValueToIntList(Integer value) {
        intList.add(value);
    }
}

```

```

    public List<Integer> getIntList() {
        return Collections.unmodifiableList(intList);
    }
}

```

なコレクションをしようとすると、がスローされます。

```

import java.util.List;

public class App {

    public static void main(String[] args) {
        MyPojoClass pojo = new MyPojoClass();
        pojo.addValueToIntList(42);

        List<Integer> list = pojo.getIntList();
        list.add(69);
    }
}

```

```

Exception in thread "main" java.lang.UnsupportedOperationException
    at java.util.Collections$UnmodifiableCollection.add(Collections.java:1055)
    at App.main(App.java:12)

```

コレクションのりし

リストをする

```

List<String> names = new ArrayList<>(Arrays.asList("Clementine", "Duran", "Mike"));

```

Java SE 8

```

names.forEach(System.out::println);

```

がな

```

names.parallelStream().forEach(System.out::println);

```

Java SE 5

```

for (String name : names) {
    System.out.println(name);
}

```

Java SE 5

```

for (int i = 0; i < names.size(); i++) {
    System.out.println(names.get(i));
}

```

Java SE 1.2

```
//Creates ListIterator which supports both forward as well as backward traversal
ListIterator<String> listIterator = names.listIterator();

//Iterates list in forward direction
while(listIterator.hasNext()){
    System.out.println(listIterator.next());
}

//Iterates list in backward direction once reaches the last element from above iterator in
forward direction
while(listIterator.hasPrevious()){
    System.out.println(listIterator.previous());
}
```

セットをする

```
Set<String> names = new HashSet<>(Arrays.asList("Clementine", "Duran", "Mike"));
```

Java SE 8

```
names.forEach(System.out::println);
```

Java SE 5

```
for (Iterator<String> iterator = names.iterator(); iterator.hasNext(); ) {
    System.out.println(iterator.next());
}

for (String name : names) {
    System.out.println(name);
}
```

Java SE 5

```
Iterator iterator = names.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

マップをする

```
Map<Integer, String> names = new HashMap<>();
names.put(1, "Clementine");
names.put(2, "Duran");
names.put(3, "Mike");
```

Java SE 8

```
names.forEach((key, value) -> System.out.println("Key: " + key + " Value: " + value));
```


Java SE 5

```
for (Map.Entry<Integer, String> entry : names.entrySet()) {
    System.out.println(entry.getKey());
    System.out.println(entry.getValue());
}

// Iterating over only keys
for (Integer key : names.keySet()) {
    System.out.println(key);
}

// Iterating over only values
for (String value : names.values()) {
    System.out.println(value);
}
```

Java SE 5

```
Iterator entries = names.entrySet().iterator();
while (entries.hasNext()) {
    Map.Entry entry = (Map.Entry) entries.next();
    System.out.println(entry.getKey());
    System.out.println(entry.getValue());
}
```

のコレクション

のコレクションをするのがなもあります。 `Collections` クラスは、そのようなコレクションをにするメソッドをします。

```
List<String> anEmptyList = Collections.emptyList();
Map<Integer, Date> anEmptyMap = Collections.emptyMap();
Set<Number> anEmptySet = Collections.emptySet();
```

これらのメソッドはであり、されたコレクションをりてられたににします。つまり、例えば `emptyList()` は `List` のタイプの `List` をりてられ、 `emptySet()` および `emptyMap()` にしてもりてられます。

これらのメソッドによってされるコレクションは、をするメソッド `add`、 `put` などをひそうとすると `UnsupportedOperationException` をスローするというです。これらのコレクションは、 `null` をするか、 `new` をしてオブジェクトをするのではなく、のメソッドのやそののデフォルトのとしてにちます。

コレクションとプリミティブ

Javaのコレクションはオブジェクトにしてのみします。つまり、Java `Map<int, int>` はありません。わりに、 `Map<Integer, Integer>` ようにプリミティブをオブジェクトにむがあります。Javaオートボックスにより、これらのコレクションのながになります。

```
Map<Integer, Integer> map = new HashMap<>();
map.put(1, 17); // Automatic boxing of int to Integer objects
int a = map.get(1); // Automatic unboxing.
```

ながら、このオーバーヘッドはです。 `HashMap<Integer, Integer>` は、エン트리あたり72バイト
例えば、ポインタをつ64ビットJVMで、256よりきいをし、マップの50のロードをしますをとしま
す。のデータはわずか8バイトなので、なオーバーヘッドがします。さらに、2レベルのMap
→Entry→Valueがですが、にいです。

プリミティブのなコレクションをすると、パフォーマンスがにするプリミティブデータ50ので1
エン트리あたり16バイトしかない、つまりメモリが4なく、が1レベルないのコレクションがされ
たライブラリがいくつかしますJavaの。

Iteratorをして**List**からするをする。

のでは、ループのリストからをするにきましたが、は `Iterator` インターフェースをしてなのをえ
ました。

これは、あなたがきたいリストのしたをうときにつのあるトリックのデモンストレーションで
す。

これは、ループのリストからをするにのみされます。

だから、いつものようにリストをしましょう

```
String[] names = {"James", "Smith", "Sonny", "Huckle", "Berry", "Finn", "Allan"};
List<String> nameList = new ArrayList<>();

//Create a List from an Array
nameList.addAll(Arrays.asList(names));

String[] removeNames = {"Sonny", "Huckle", "Berry"};
List<String> removeNameList = new ArrayList<>();

//Create a List from an Array
removeNameList.addAll(Arrays.asList(removeNames));
```

のメソッドは、2つのCollectionオブジェクトをり、 `nameList` とする `removeNameList` のをするとい
をし `nameList` 。

```
private static void removeNames(Collection<String> collection1, Collection<String>
collection2) {
    //get Iterator.
    Iterator<String> iterator = collection1.iterator();

    //Loop while collection has items
    while(iterator.hasNext()){
        if (collection2.contains(iterator.next()))
            iterator.remove(); //remove the current Name or Item
    }
}
```

メソッドをびし、 `nameList` と `removeNameList` をのように `removeNames(nameList, removeNameList);`
のがされます。

をするのリスト **James Smith Sonny Huckle Berry Finn Allan**

をしたのリスト **James Smith Finn Allan**

リストのりしをするのにな、コレクションのない。

Iteratorまたは**for-each**ループでするの**Iterable**をする。

イテレータまたはfor-eachループをしてコレクションをにできるようにするには、のをするがあります。

1. たちがりしたいものは**Iterable**で `iterator()` をする **Iterable**あり `iterator()` 。
2. `hasNext()` 、 `next()` および `remove()` オーバーライドして `java.util.Iterator` します。

のエンティティをしてリンクリストをにするなリンクリストのをしました。

```
package org.algorithms.linkedlist;

import java.util.Iterator;
import java.util.NoSuchElementException;

public class LinkedList<T> implements Iterable<T> {

    Node<T> head, current;

    private static class Node<T> {
        T data;
        Node<T> next;

        Node(T data) {
            this.data = data;
        }
    }

    public LinkedList(T data) {
        head = new Node<>(data);
    }

    public Iterator<T> iterator() {
        return new LinkedListIterator();
    }

    private class LinkedListIterator implements Iterator<T> {

        Node<T> node = head;

        @Override
        public boolean hasNext() {
            return node != null;
        }

        @Override
        public T next() {
            if (!hasNext())
                throw new NoSuchElementException();
            Node<T> prevNode = node;
            node = node.next;
            return prevNode.data;
        }
    }
}
```

```

    }

    @Override
    public void remove() {
        throw new UnsupportedOperationException("Removal logic not implemented.");
    }
}

public void add(T data) {
    Node current = head;
    while (current.next != null)
        current = current.next;
    current.next = new Node<>(data);
}

}

class App {
    public static void main(String[] args) {

        LinkedList<Integer> list = new LinkedList<>(1);
        list.add(2);
        list.add(4);
        list.add(3);

        //Test #1
        System.out.println("using Iterator:");
        Iterator<Integer> itr = list.iterator();
        while (itr.hasNext()) {
            Integer i = itr.next();
            System.out.print(i + " ");
        }

        //Test #2
        System.out.println("\n\nusing for-each:");
        for (Integer data : list) {
            System.out.print(data + " ");
        }
    }
}

```

```

using Iterator:
1 2 4 3
using for-each:
1 2 4 3

```

これはJava 7でします。Java 5とJava 6でもできるようにするには、のものをきえます。

```
LinkedList<Integer> list = new LinkedList<>(1);
```

と

```
LinkedList<Integer> list = new LinkedList<Integer>(1);
```

のあるをみむことによって、のバージョンにすることができます。

とし

これは、イテレータオブジェクトによってされたメソッドのメソッドをしてコレクションをしているにコレクションがされたにします。たとえば、のリストがあり、たぶのあるものをすべてしたいとします。

```
List<IHat> hats = new ArrayList<>();
hats.add(new Ushanka()); // that one has ear flaps
hats.add(new Fedora());
hats.add(new Sombrero());
for (IHat hat : hats) {
    if (hat.hasEarFlaps()) {
        hats.remove(hat);
    }
}
```

このコードをすると、コードがにコレクションをするため、**ConcurrentModificationException**がします。じリストでしているのスレッドのうちの1つがコレクションをしようとしているのにのスレッドがしようとしている、じがすることがあります。のスレッドでのコレクションのはなことです。ロック、にされたなコレクション、からクローンされたコレクションのなど、プログラミングツールボックスののツールでするがあります。

サブコレクション

リスト **subList** **int fromIndex**、**int toIndex**

ここから**fromIndex**はインクルーシブであり、**toIndex**はです。

```
List list = new ArrayList();
List list1 = list.subList(fromIndex,toIndex);
```

1. されたにリストがしないは、**IndexOutOfBoundsException**をスローします。
2. リスト1にえられたは、リストのじにします。これは、コレクションとばれます。
3. **fromIndex**が**toIndex****fromIndex**> **toIndex**よりきい、**IllegalArgumentException**がスローされます。

```
List<String> list = new ArrayList<String>();
List<String> list = new ArrayList<String>();
list.add("Hello1");
list.add("Hello2");
System.out.println("Before Sublist "+list);
List<String> list2 = list.subList(0, 1);
list2.add("Hello3");
System.out.println("After sublist changes "+list);
```

サブリストの[Hello1、Hello2]

サブリストの[Hello1、Hello3、Hello2]

サブセットをするfromIndex、toIndex

ここからfromIndexはインクルーシブであり、toIndexはです。

```
Set set = new TreeSet();  
Set set1 = set.subSet(fromIndex,toIndex);
```

されたセットは、そののをしようとするときIllegalArgumentExceptionをスローします。

マップサブマップfromKey、toKey

fromKeyはインクルーシブであり、toKeyはです。

```
Map map = new TreeMap();  
Map map1 = map.get(fromKey,toKey);
```

fromKeyがtoKeyより大きい、またはこのマップがされたをもち、fromKeyまたはtoKeyがのにあるは、IllegalArgumentExceptionをスローします。

すべてのコレクションはバックアップされたコレクションをサポートします。これは、サブコレクションにえられたがメインコレクションでじをうことをします。

オンラインでコレクションをむ <https://riptutorial.com/ja/java/topic/90/コレクション>

91: コレクションの

き

Javaにはさまざまなコレクションがされています。するコレクションをするのはいいことがあります。ジョブのいいコレクションをするためのなフローチャートについては、「」セクションをしてください。

Examples

Java コレクションのフローチャート

のフローチャートをして、ジョブのいいコレクションをします。

このフローチャートは[<http://i.stack.imgur.com/aSDsG.png>]についています。

オンラインでコレクションのをむ <https://riptutorial.com/ja/java/topic/10846/> コレクションの

92: コレクションファクトリメソッド

き

Java 9のにより、JavaのCollections APIにくのがもたらされます。その1つはコレクションファクトリメソッドです。これらのメソッドをすると、でもでも、 のコレクションをにできます。

これらのファクトリメソッドは、 `List<E>`、 `Set<E>`、 および `Map<K, V>`のインタフェースでのみできます。

- `static <E> List<E> of()`
- `static <E> List<E> of(E e1)`
- `static <E> List<E> of(E e1, E e2)`
- `static <E> List<E> of(E e1, E e2, ..., E e9, E e10)`
- `static <E> List<E> of(E... elements)`
- `static <E> Set<E> of()`
- `static <E> Set<E> of(E e1)`
- `static <E> Set<E> of(E e1, E e2)`
- `static <E> Set<E> of(E e1, E e2, ..., E e9, E e10)`
- `static <E> Set<E> of(E... elements)`
- `static <K,V> Map<K,V> of()`
- `static <K,V> Map<K,V> of(K k1, V v1)`
- `static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2)`
- `static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, ..., K k9, V v9, K k10, V v10)`
- `static <K,V> Map<K,V> ofEntries(Map.Entry<? extends K,? extends V>... entries)`

パラメーター

メソッドw/パラメータ	
<code>List.of(E e)</code>	クラスまたはインタフェースになることができるジェネリック。
<code>Set.of(E e)</code>	クラスまたはインタフェースになることができるジェネリック。
<code>Map.of(K k, V v)</code>	それぞれがクラスまたはインタフェースになることができるジェネリックのキーとのペア。
<code>Map.of(Map.Entry<? extends K, ? extends V> entry)</code>	そのキーが K かそののいずれかであり、そのが V かそのである <code>Map.Entry</code> インスタンス。

Examples

リストファクトリメソッドの

- `List<Integer> immutableEmptyList = List.of();`
◦ のな `List<Integer>` します。
- `List<Integer> immutableList = List.of(1, 2, 3, 4, 5);`
◦ の `List<Integer>` を5つのでします。
- `List<Integer> mutableList = new ArrayList<>(immutableList);`
◦ な `List<Integer>` の `List<Integer>` 。

セットファクトリメソッドの

- `Set<Integer> immutableEmptySet = Set.of();`
◦ の `Set<Integer>` します。
- `Set<Integer> immutableSet = Set.of(1, 2, 3, 4, 5);`
◦ 5つのでの `Set<Integer>` をします。
- `Set<Integer> mutableSet = new HashSet<>(immutableSet);`
◦ `Set<Integer>` の `Set<Integer>` からな `Set<Integer>` します。

ファクトリメソッドの

- `Map<Integer, Integer> immutableEmptyMap = Map.of();`
◦ のの `Map<Integer, Integer>` します。
- `Map<Integer, Integer> immutableMap = Map.of(1, 2, 3, 4);`
◦ 2つのキーエントリをしての `Map<Integer, Integer>` をします。
- `Map<Integer, Integer> immutableMap = Map.ofEntries(Map.entry(1, 2), Map.entry(3, 4));`
◦ 2つのキーエントリをしての `Map<Integer, Integer>` をします。
- `Map<Integer, Integer> mutableMap = new HashMap<>(immutableMap);`
◦ の `Map<Integer, Integer>` からな `Map<Integer, Integer>` をします。

オンラインでコレクションファクトリメソッドをむ <https://riptutorial.com/ja/java/topic/9783/コレクションファクトリメソッド>

93: コンカレントコレクション

き

は、にのスレッドによるアクセスをする[コレクション][1]です。なるスレッドは、コレクションのをりしし、をまたはします。コレクションは、コレクションがしないようにするがあります。
。 [1]<http://stackoverflow.com/documentation/java/90/collections#t=201612221936497298484>

Examples

スレッドセーフなコレクション

デフォルトでは、さまざまなCollectionはスレッドセーフではありません。

しかし、コレクションをスレッドセーフにすることはかなりです。

```
List<String> threadSafeList = Collections.synchronizedList(new ArrayList<String>());
Set<String> threadSafeSet = Collections.synchronizedSet(new HashSet<String>());
Map<String, String> threadSafeMap = Collections.synchronizedMap(new HashMap<String, String>());
```

スレッドセーフなコレクションをするときは、スレッドセーフなラッパーをしてのみ、のコレクションからアクセスしないでください。

Java SE 5

Java 5、`java.util.collections`は、さまざまな`Collections.synchronized`メソッドがないスレッドセーフなコレクションがいくつかあります。

```
List<String> threadSafeList = new CopyOnWriteArrayList<String>();
Set<String> threadSafeSet = new ConcurrentHashSet<String>();
Map<String, String> threadSafeMap = new ConcurrentHashMap<String, String>();
```

コンカレントコレクション

コレクションは、スレッドセーフなコレクションのであり、でのいをしします。

スレッドセーフなコレクションはのスレッドからにをまたはできますが、ずしもじコンテキストでなをうはありません1つのスレッドでコレクションをにできないがあります。をする。

これは、がされるです。

りしは`addAll`、`removeAll`、またはコレクションのコピーコンストラクタやそののによる、べえなど、コレクションのいくつかのバルクメソッドのであることがため、コレクションのユースケースはかなりきいです。

たとえば、Java SE 5 `java.util.concurrent.CopyOnWriteArrayList` スレッドセーフとである `List` の、その [Javadoc](#) はべています

スナップショットスタイルのイテレータメソッドは、イテレータがされたでのへのをします。これは、イテレータのはされないため、はであり、イテレータは `ConcurrentModificationException` をスローしないことがされています。

したがって、のコードはです。

```
public class ThreadSafeAndConcurrent {

    public static final List<Integer> LIST = new CopyOnWriteArrayList<>();

    public static void main(String[] args) throws InterruptedException {
        Thread modifier = new Thread(new ModifierRunnable());
        Thread iterator = new Thread(new IteratorRunnable());
        modifier.start();
        iterator.start();
        modifier.join();
        iterator.join();
    }

    public static final class ModifierRunnable implements Runnable {
        @Override
        public void run() {
            try {
                for (int i = 0; i < 50000; i++) {
                    LIST.add(i);
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

    public static final class IteratorRunnable implements Runnable {
        @Override
        public void run() {
            try {
                for (int i = 0; i < 10000; i++) {
                    long total = 0;
                    for(Integer inList : LIST) {
                        total += inList;
                    }
                    System.out.println(total);
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

にするもう1つののは [ConcurrentLinkedQueue](#) です。

イテレータはくしており、イテレータのまたはのあるでキューのをするをします。らは `java.util.ConcurrentModificationException` をげず、のとしてめることができます。イ

テレータのにキューにまれるは、だけされます。

コレクションがしているかどうかをするには、javadocをチェックするがあります。 `iterator()` メソッド "fail fast"、 "weakly consistent"など `iterator()` によってされるイテレータのは、もなです。

スレッドセーフであるかの

のコードでは、 `LIST`を

```
public static final List<Integer> LIST = Collections.synchronizedList(new ArrayList<>());
```

のマルチCPU /コアアーキテクチャではにがするがあります。

`Collections`ユーティリティメソッドからのされた`Collections`は、の/ではスレッドセーフですが、りしはできませんそれにされるのコレクションはにしているをきます。

ConcurrentHashMapへの

```
public class InsertIntoConcurrentHashMap
{
    public static void main(String[] args)
    {
        ConcurrentHashMap<Integer, SomeObject> concurrentHashMap = new ConcurrentHashMap<>();

        SomeObject value = new SomeObject();
        Integer key = 1;

        SomeObject previousValue = concurrentHashMap.putIfAbsent(1, value);
        if (previousValue != null)
        {
            //Then some other value was mapped to key = 1. 'value' that was passed to
            //putIfAbsent method is NOT inserted, hence, any other thread which calls
            //concurrentHashMap.get(1) would NOT receive a reference to the 'value'
            //that your thread attempted to insert. Decide how you wish to handle
            //this situation.
        }
        else
        {
            //'value' reference is mapped to key = 1.
        }
    }
}
```

オンラインでコンカレントコレクションをむ <https://riptutorial.com/ja/java/topic/8363/コンカレントコレクション>

94: コンストラクタ

き

ではありませんが、Javaのコンストラクタは、オブジェクトのになクラスのをインスタンスするためにコンパイラによってされるメソッドです。このトピックでは、Javaクラス・コンストラクターのなをします。

Javaでは、コンストラクタのセマンティクスのなについて、らくしています。それらは[JLS§8.8](#)でつけることができます

Examples

デフォルトコンストラクタ

コンストラクタの"デフォルト"はがないことです。あなたがのコンストラクタをしない、コンパイラはあなたのためのデフォルトコンストラクタをします。これは、の2つのスニペットがにであることをします。

```
public class TestClass {  
    private String test;  
}
```

```
public class TestClass {  
    private String test;  
    public TestClass() {  
  
    }  
}
```

デフォルトコンストラクタのは、クラスのとじです。したがって、クラスされたパッケージはプライベートにパッケージプライベートのデフォルトのコンストラクタをとっています

しかし、デフォルトのコンストラクタをしている、コンパイラはデフォルトのコンストラクタをしません。したがって、これらはではありません

```
public class TestClass {  
    private String test;  
    public TestClass(String arg) {  
    }  
}
```

```
public class TestClass {  
    private String test;  
    public TestClass() {  
    }  
    public TestClass(String arg) {  
    }  
}
```

```
    }  
}
```

されたコンストラクタはをしないことにしてください。つまり、がないり、クラスのすべてのフィールドにデフォルトがされます。

```
public class TestClass {  
  
    private String testData;  
  
    public TestClass() {  
        testData = "Test"  
    }  
}
```

コンストラクターはのようにはびされます。

```
TestClass testClass = new TestClass();
```

をつコンストラクタ

コンストラクターはののきでできます。

```
public class TestClass {  
  
    private String testData;  
  
    public TestClass(String testData) {  
        this.testData = testData;  
    }  
}
```

このようにびされた

```
TestClass testClass = new TestClass("Test Data");
```

クラスは、なるシグネチャをつのコンストラクタをつことができます。コンストラクタびしインスタンスにじクラスののコンストラクタをびすをさせるには、`this()` します。

```
public class TestClass {  
  
    private String testData;  
  
    public TestClass(String testData) {  
        this.testData = testData;  
    }  
  
    public TestClass() {  
        this("Test"); // testData defaults to "Test"  
    }  
}
```

このようにびされた

```
TestClass testClass1 = new TestClass("Test Data");
TestClass testClass2 = new TestClass();
```

コンストラクタをびす

ParentクラスとChildクラスがあるとします。Childインスタンスをするには、に、ChildコンストラクタのにgebinningでされるいくつかのParentコンストラクタがです。のコンストラクタとしてなをしてsuper(...)をにびして、なParentコンストラクタをできます。これにより、Childクラスのコンストラクタでじコードをきえるのではなく、Parentクラスのコンストラクタをしてをできます。

super(...) メソッド なし

のうちに、なしのバージョンsuper()はにえないものとされます

```
class Parent {
    private String name;
    private int age;

    public Parent() {} // necessary because we call super() without arguments

    public Parent(String tName, int tAge) {
        name = tName;
        age = tAge;
    }
}

// This does not even compile, because name and age are private,
// making them invisible even to the child class.
class Child extends Parent {
    public Child() {
        // compiler implicitly calls super() here
        name = "John";
        age = 42;
    }
}
```

super() メソッド

```
class Parent {
    private String name;
    private int age;
    public Parent(String tName, int tAge) {
        name = tName;
        age = tAge;
    }
}

class Child extends Parent {
    public Child() {
        super("John", 42); // explicit super-call
    }
}
```

```
}
```

のコンストラクタまたはスーパーコンストラクタへのびしは、コンストラクタなのでなければなりません**MUST**。

`super(...)` コンストラクターをにびすと、するコンストラクターがするがありますこれはですが、そうではありません。

`super(...)` コンストラクタをにびさない、クラスにはなしのコンストラクタがです。クラスがしないはコンパイラによってにされるか、デフォルトとしてされますのコンストラクタ。

```
class Parent{
    public Parent(String tName, int tAge) {}
}

class Child extends Parent{
    public Child(){}
}
```

クラスParentにはデフォルトのコンストラクタがないため、コンパイラはコンストラクタに`super`をできません。このコードはコンパイルされません。あなたはのにうようにコンストラクタをしなければなりません、あるいはあなたの`super`コールをのようにくがあります

```
class Child extends Parent{
    public Child(){
        super("", 0);
    }
}
```

オンラインでコンストラクタをむ <https://riptutorial.com/ja/java/topic/682/コンストラクタ>

95: コンソールI/O

Examples

コンソールからユーザーをみる

BufferedReader

```
System.out.println("Please type your name and press Enter.");

BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
try {
    String name = reader.readLine();
    System.out.println("Hello, " + name + "!");
} catch(IOException e) {
    System.out.println("An error occurred: " + e.getMessage());
}
```

このコードには、のインポートが必要です。

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
```

Scanner

Java SE 5

```
System.out.println("Please type your name and press Enter");

Scanner scanner = new Scanner(System.in);
String name = scanner.nextLine();

System.out.println("Hello, " + name + "!");
```

このでは、のインポートが必要です。

```
import java.util.Scanner;
```

のをみるには、 scanner.nextLine() りしびし scanner.nextLine()。

```
System.out.println("Please enter your first and your last name, on separate lines.");

Scanner scanner = new Scanner(System.in);
String firstName = scanner.nextLine();
String lastName = scanner.nextLine();

System.out.println("Hello, " + firstName + " " + lastName + "!");
```

`Strings nextLine()`、`next()`および`nextLine()`をするは2つあります。`next()`はのスペース「トークン」ともばれるまでテキストをし、`nextLine()`はユーザーがenterをすまでしたすべてのテキストをします。

`Scanner`は、`String`のデータを見るためのユーティリティメソッドもします。これらには、

```
scanner.nextByte();
scanner.nextShort();
scanner.nextInt();
scanner.nextLong();
scanner.nextFloat();
scanner.nextDouble();
scanner.nextBigInteger();
scanner.nextBigDecimal();
```

これらののいずれかをける`has`と`hasNextLine()`、`hasNextInt()`をし`true`ストリームがタイプののを持っている。これらのメソッドは、がされたでないたとえば、`nextInt()` "a"をするなど、プログラムをクラッシュさせます。これをぐために`try {} catch() {}`をうことができます [を](#)

```
Scanner scanner = new Scanner(System.in); //Create the scanner
scanner.useLocale(Locale.US); //Set number format excepted
System.out.println("Please input a float, decimal separator is .");
if (scanner.hasNextFloat()){ //Check if it is a float
    float fValue = scanner.nextFloat(); //retrive the value directly as float
    System.out.println(fValue + " is a float");
}else{
    String sValue = scanner.next(); //We can not retrive as float
    System.out.println(sValue + " is not a float");
}
```

System.console

Java SE 6

```
String name = System.console().readLine("Please type your name and press Enter%n");

System.out.printf("Hello, %s!", name);

//To read passwords (without echoing as in unix terminal)
char[] password = System.console().readPassword();
```

- みみがされている
- をできます

これは、プログラムがのコマンドラインからされ、ストリームをリダイレクトしないにのみします。プログラムがEclipseなどのIDEからされているときはしません。IDEとストリームリダイレクトですコードについては、のをしてください。

なコマンドラインの

なプロトタイプやなコマンドラインのについては、のループがです。

```
public class ExampleCli {

    private static final String CLI_LINE    = "example-cli>"; //console like string

    private static final String CMD_QUIT    = "quit";        //string for exiting the program
    private static final String CMD_HELLO   = "hello";        //string for printing "Hello World!"
on the screen
    private static final String CMD_ANSWER  = "answer";        //string for printing 42 on the
screen

    public static void main(String[] args) {
        ExampleCli claimCli = new ExampleCli();    // creates an object of this class

        try {
            claimCli.start();    //calls the start function to do the work like console
        }
        catch (IOException e) {
            e.printStackTrace();    //prints the exception log if it is failed to do get the
user input or something like that
        }
    }

    private void start() throws IOException {
        String cmd = "";

        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        while (!cmd.equals(CMD_QUIT)) {    // terminates console if user input is "quit"
            System.out.print(CLI_LINE);    //prints the console-like string

            cmd = reader.readLine();    //takes input from user. user input should be started
with "hello", "answer" or "quit"
            String[] cmdArr = cmd.split(" ");

            if (cmdArr[0].equals(CMD_HELLO)) {    //executes when user input starts with
"hello"
                hello(cmdArr);
            }
            else if (cmdArr[0].equals(CMD_ANSWER)) {    //executes when user input starts with
"answer"
                answer(cmdArr);
            }
        }

        // prints "Hello World!" on the screen if user input starts with "hello"
        private void hello(String[] cmdArr) {
            System.out.println("Hello World!");
        }

        // prints "42" on the screen if user input starts with "answer"
        private void answer(String[] cmdArr) {
            System.out.println("42");
        }
    }
}
```

コンソールでの

メソッド `PrintWriter.format` `System.out.format` をじてびされますをして、したをコンソールに `System.out.format` することができます。このメソッドは、フォーマットとフォーマットするのオブジェクトをむ `String` をけります。

```
String rowsStrings[] = new String[] { "1",
                                         "1234",
                                         "1234567",
                                         "123456789"};

String column1Format = "%-3s";    // min 3 characters, left aligned
String column2Format = "%-5.8s";  // min 5 and max 8 characters, left aligned
String column3Format = "%6.6s";   // fixed size 6 characters, right aligned
String formatInfo = column1Format + " " + column2Format + " " + column3Format;

for(int i = 0; i < rowsStrings.length; i++) {
    System.out.format(formatInfo, rowsStrings[i], rowsStrings[i], rowsStrings[i]);
    System.out.println();
}
```

```
1    1        1
1234 1234    1234
1234567 1234567 123456
123456789 12345678 123456
```

サイズのをすると、をサイズのののようなですることができます。

```
String rowsStrings[] = new String[] { "1",
                                         "1234",
                                         "1234567",
                                         "123456789"};

String column1Format = "%-3.3s";  // fixed size 3 characters, left aligned
String column2Format = "%-8.8s";  // fixed size 8 characters, left aligned
String column3Format = "%6.6s";   // fixed size 6 characters, right aligned
String formatInfo = column1Format + " " + column2Format + " " + column3Format;

for(int i = 0; i < rowsStrings.length; i++) {
    System.out.format(formatInfo, rowsStrings[i], rowsStrings[i], rowsStrings[i]);
    System.out.println();
}
```

```
1    1        1
123 1234    1234
123 1234567 123456
123 12345678 123456
```

フォーマットの

- `%s` フォーマットされていない
- `%5s` を 5でフォーマットします。がいには、5つのとにパディングされます
- `%-5s` を 5でフォーマットします。がいには、5つのにパディングされ、めされます

- `%5.10s` 5 10までののを、が5よりいには、5つのと、えにめられます。が10よりもいには、10のえにりてられます
- `%-5.5s` を5でフォーマットしますとしい。が5よりいには、5つのにパディングされ、めされます。が5よりもいには、5つのにりてられ、めされます

オンラインでコンソールI/Oをむ <https://riptutorial.com/ja/java/topic/126/コンソールj---o>

96: ジェネリックス

き

ジェネリックスは、コンパイルののをしなから、またはメソッドがさまざまなオブジェクトをできるように、Javaのシステムをするプログラミングのです。に、Javaコレクション・フレームワークは、ジェネリックスをサポートして、コレクション・インスタンスにされているオブジェクトのタイプをします。

- クラス `ArrayList<E> {}` //パラメータEをつクラス
- クラス `HashMap<K, V> {}` // 2つのパラメータKとVをつクラス
- `<E> void printE element{}` //パラメータEをつメソッド
- `ArrayList<String>`のです。 //ジェネリックスクラスの
- `ArrayList<>`オブジェクト。 //パラメータがなクラスの
- 新しい`ArrayList<String>`//クラスのインスタンス
- 新しい`ArrayList<>`//「diamond」Java 7をしたインスタンス

ジェネリックスは、タイプによってJavaでされます。つまり、にジェネリックスクラスのインスタンスでされたタイプはできません。たとえば、`List<String> names = new ArrayList<>();`というステートメントがあり`List<String> names = new ArrayList<>();`にエレメントタイプ`String`をできないリストオブジェクトをします。ただし、リストが`List<String>`のフィールドにされている、またはこののメソッド/コンストラクタパラメータにされた、またはそののメソッドからされた、にをできますJava Reflection APIをします。

これは、ジェネリックスたとえば `(List<String>) list` にキャストするときに、キャストがチェックされていないキャストであることもします。パラメータ`<String>`がされるため、JVMは`List<String>`から`List<String>`へのキャストが新しい`List<?>`することはできません。JVMはに`List to List`キャストしかることができません。

Examples

クラスの

ジェネリックスをすると、クラス、インタフェース、およびメソッドは、のクラスとインタフェースをパラメータとしてできます。

このでは、ジェネリックスクラス`Param`をして、カッコ `<>` でられたのパラメータ `T`をとります。

```
public class Param<T> {
    private T value;

    public T getValue() {
        return value;
    }
}
```

```

    }

    public void setValue(T value) {
        this.value = value;
    }
}

```

このクラスをインスタンスするには、`T`わりにをします。たとえば、`Integer`

```
Param<Integer> integerParam = new Param<Integer>();
```

`type`は、やそののをものになります。

```

Param<String[]> stringArrayParam;
Param<int[][]> int2dArrayParam;
Param<Param<Object>> objectNestedParam;

```

Java SE 7では、`type`は、**ダイヤモンド**と呼ばれる `<>` ののセットでえることができます。

Java SE 7

```
Param<Integer> integerParam = new Param<>();
```

のとはなり、パラメータにはのはありません。しかし、らののは、のののです。これはな
JavaDocsでもてはまります。

としては、**"type"**のは`T`、**"element"**のは`E`、**"key" / "value"**のは`K / V`があります。

ジェネリッククラスの

```

public abstract class AbstractParam<T> {
    private T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}

```

`AbstractParam`は、パラメータ`T`された**クラス**です。このクラスをするとき、そのパラメータは`<>`にかれたでえることができます。そうでないは、パラメータはされません。の1および2のでは、`String`と`Integer`が`type`パラメータをきえます。3のでは、`type`パラメータはされません。4のではジェネリックはまったくしないため、クラスに`Object`パラメータがあるとています。コンパイラは`AbstractParam`がのであることをしますが、`ObjectParam`クラスをコンパイルします。5のは、2つのパラメータの「パラメータ」ををち、2のパラメータをスーパークラスにされるパラメータ

としてします。

```
public class Email extends AbstractParam<String> {
    // ...
}

public class Age extends AbstractParam<Integer> {
    // ...
}

public class Height<T> extends AbstractParam<T> {
    // ...
}

public class ObjectParam extends AbstractParam {
    // ...
}

public class MultiParam<T, E> extends AbstractParam<E> {
    // ...
}
```

はのとおりで。

```
Email email = new Email();
email.setValue("test@example.com");
String retrievedEmail = email.getValue();

Age age = new Age();
age.setValue(25);
Integer retrievedAge = age.getValue();
int autounboxedAge = age.getValue();

Height<Integer> heightInInt = new Height<>();
heightInInt.setValue(125);

Height<Float> heightInFloat = new Height<>();
heightInFloat.setValue(120.3f);

MultiParam<String, Double> multiParam = new MultiParam<>();
multiParam.setValue(3.3);
```

Email クラスでは、`T getValue()` メソッドは `String getValue()` シグネチャをつかのようにし、`void setValue(T)` メソッドは `void setValue(String)` としてされているかのようにし `void setValue(String)` 。

の `{}` でのクラスでインスタンスすることもできます。

```
AbstractParam<Double> height = new AbstractParam<Double>(){};
height.setValue(198.6);
```

のクラスをつダイヤモンドをすることはできません。

のタイプのパラメータ

Javaでは、ジェネリッククラスまたはインターフェイスでのパラメータをできます。のパラメータは、クラスまたはインタフェースで、のにカンマでったタイプのリストをすることによってできます。

```
public class MultiGenericParam<T, S> {
    private T firstParam;
    private S secondParam;

    public MultiGenericParam(T firstParam, S secondParam) {
        this.firstParam = firstParam;
        this.secondParam = secondParam;
    }

    public T getFirstParam() {
        return firstParam;
    }

    public void setFirstParam(T firstParam) {
        this.firstParam = firstParam;
    }

    public S getSecondParam() {
        return secondParam;
    }

    public void setSecondParam(S secondParam) {
        this.secondParam = secondParam;
    }
}
```

はのとおりで。

```
MultiGenericParam<String, String> aParam = new MultiGenericParam<String, String>("value1",
"value2");
MultiGenericParam<Integer, Double> dayOfWeekDegrees = new MultiGenericParam<Integer,
Double>(1, 2.6);
```

ジェネリックメソッドの

メソッドにはジェネリックのパラメータもあります。

```
public class Example {

    // The type parameter T is scoped to the method
    // and is independent of type parameters of other methods.
    public <T> List<T> makeList(T t1, T t2) {
        List<T> result = new ArrayList<T>();
        result.add(t1);
        result.add(t2);
        return result;
    }
}
```

```

public void usage() {
    List<String> listString = makeList("Jeff", "Atwood");
    List<Integer> listInteger = makeList(1, 2);
}
}

```

のをメソッドにすはしないことにしてください。コンパイラは、ターゲットをするなどまたはののについて、をします。には、コールタイプをしいものにするもなタイプのをします。

まれに、なでこのをオーバーライドするがあります。

```

void usage() {
    consumeObjects(this.<Object>makeList("Jeff", "Atwood").stream());
}

void consumeObjects(Stream<Object> stream) { ... }

```

このでは、コンパイラは、`stream()`をびしたに`Object`が`T`にであり、そうでなければ`makeList`について`String`をすることを「み」できないためです。Javaは、メソッドがびされたクラスまたはオブジェクトサポートしていないことにしてください`this`がにされる、のでは。

ダイヤモンド

Java SE 7

Java 7はDiamond¹をして、ジェネリッククラスのインスタンス化についていくつかのボイラープレートをしました。Java 7+をすると、のようにくことができます。

```
List<String> list = new LinkedList<>();
```

のバージョンでかなければならなかったはのとおりです。

```
List<String> list = new LinkedList<String>();
```

1つのはクラスであり、インスタンスでパラメータをするがあります。

```

// This will compile:

Comparator<String> caseInsensitiveComparator = new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return s1.compareToIgnoreCase(s2);
    }
};

// But this will not:

Comparator<String> caseInsensitiveComparator = new Comparator<>() {
    @Override
    public int compare(String s1, String s2) {

```

```

        return s1.compareToIgnoreCase(s2);
    }
};

```

Java SE 8

[Anonymous Inner Classes](#)でのダイヤモンドのはJava 7および8ではサポートされていませんが、[Java 9](#)のとしてまれます。

1 - <>を「ダイヤモンド」とぶもいます。これはっています。ダイヤモンドはとしてしません。JLSやオペレータとしてののJavaチュートリアルのだこにもされていません。かに、<>はのJavaトークンでもありません。それはむしろ<トークンのに>トークンであり、2つのにやコメントをれることはですスタイルですが。JLSとチュートリアルでは、<>は「ダイヤモンド」としてしています。したがって、しいです。

のをとする「ABを」

のをするには、ジェネリックをとすることができます。

のリストをソートしたいが、NumberはComparableしていない。

```

public <T extends Number & Comparable<T>> void sortNumbers( List<T> n ) {
    Collections.sort( n );
}

```

このでは、TはNumberをし、IntegerまたはBigDecimalようなすべての「の」みみのにしなければならぬが、Striped64のようなよりエキゾチックなものにはしないComparable<T>をComparable<T>があります。

のはされていないため、で1つのクラスをバインドとしてすることができ、にリストされているがあります。たとえば、<T extends Comparable<T> & Number>はされません。これは、Comparableはクラスではなくインターフェースだからです。

バインドされたクラスの

クラスでされているなは、そのをクラスでむことでできます。のなをえます。

```

public abstract class Animal {
    public abstract String getSound();
}

public class Cat extends Animal {
    public String getSound() {
        return "Meow";
    }
}

public class Dog extends Animal {
    public String getSound() {
        return "Woof";
    }
}

```

```
}
```

ジェネリックスがなければ、ジェネリックであり、がであることを持っているコンテナクラスをすることはできません

```
public class AnimalContainer<T> {

    private Collection<T> col;

    public AnimalContainer() {
        col = new ArrayList<T>();
    }

    public void add(T t) {
        col.add(t);
    }

    public void printAllSounds() {
        for (T t : col) {
            // Illegal, type T doesn't have makeSound()
            // it is used as an java.lang.Object here
            System.out.println(t.makeSound());
        }
    }
}
```

クラスでジェネリックバインドをすると、これがになりました。

```
public class BoundedAnimalContainer<T extends Animal> { // Note bound here.

    private Collection<T> col;

    public BoundedAnimalContainer() {
        col = new ArrayList<T>();
    }

    public void add(T t) {
        col.add(t);
    }

    public void printAllSounds() {
        for (T t : col) {
            // Now works because T is extending Animal
            System.out.println(t.makeSound());
        }
    }
}
```

これは、ジェネリックのなインスタンスもします。

```
// Legal
AnimalContainer<Cat> a = new AnimalContainer<Cat>();

// Legal
AnimalContainer<String> a = new AnimalContainer<String>();
```

```
// Legal because Cat extends Animal
BoundedAnimalContainer<Cat> b = new BoundedAnimalContainer<Cat>();

// Illegal because String doesn't extend Animal
BoundedAnimalContainer<String> b = new BoundedAnimalContainer<String>();
```

「T」、`「スーパーT」`、`「T」`をする

Javaジェネリックのためのは、のをすワイルドカードをバインドしてい?のとおりです。

- `? extends T`はのあるワイルドカードをします。のは、Tのサブタイプでなければならない、またはTをします。
- `? super T`はのワイルドカードをします。のは、Tのスーパータイプでなければならない、またはTのをします。

として、あなたは

- `? extends T` "みり"アクセス ""だけがな `? extends T`
- `? super T` あなたが "きみ"アクセス ""をとする、 `? super T`
- あなたがをとするならば `T` "modify"

するには、`extends`たり`super`、あなたがをしてくださいますよう、それはサブタイプとスーパータイプののがのようにあなたのコードはよりになりますので、よりいです。

```
class Shoe {}
class IPhone {}
interface Fruit {}
class Apple implements Fruit {}
class Banana implements Fruit {}
class GrannySmith extends Apple {}

public class FruitHelper {

    public void eatAll(Collection<? extends Fruit> fruits) {}

    public void addApple(Collection<? super Apple> apples) {}

}
```

コンパイラはなをできるようにします

```
public class GenericsTest {
    public static void main(String[] args){
        FruitHelper fruitHelper = new FruitHelper() ;
        List<Fruit> fruits = new ArrayList<Fruit>();
        fruits.add(new Apple()); // Allowed, as Apple is a Fruit
        fruits.add(new Banana()); // Allowed, as Banana is a Fruit
        fruitHelper.addApple(fruits); // Allowed, as "Fruit super Apple"
        fruitHelper.eatAll(fruits); // Allowed

        Collection<Banana> bananas = new ArrayList<>();
        bananas.add(new Banana()); // Allowed
        //fruitHelper.addApple(bananas); // Compile error: may only contain Bananas!
```

```

fruitHelper.eatAll(bananas); // Allowed, as all Bananas are Fruits

Collection<Apple> apples = new ArrayList<>();
fruitHelper.addApple(apples); // Allowed
apples.add(new GrannySmith()); // Allowed, as this is an Apple
fruitHelper.eatAll(apples); // Allowed, as all Apples are Fruits.

Collection<GrannySmith> grannySmithApples = new ArrayList<>();
fruitHelper.addApple(grannySmithApples); //Compile error: Not allowed.
// GrannySmith is not a supertype of Apple
apples.add(new GrannySmith()); //Still allowed, GrannySmith is an Apple
fruitHelper.eatAll(grannySmithApples); //Still allowed, GrannySmith is a Fruit

Collection<Object> objects = new ArrayList<>();
fruitHelper.addApple(objects); // Allowed, as Object super Apple
objects.add(new Shoe()); // Not a fruit
objects.add(new iPhone()); // Not a fruit
//fruitHelper.eatAll(objects); // Compile error: may contain a Shoe, too!
}

```

の T ぶ? $\text{super } T$ か? $\text{extends } T$ サブタイプでのをにするためには、 $? \text{extends } T$ がです。コンパイラはのをできます。にするとキャストするはありませんタイプセーフではなく、プログラミングエラーのとなります。

しにくいのは、**PECS**のルールをえておいてください。

Producerは「**E**xends」をし、**C**onsumerは「**S**uper」をします。

プロデューサにはきみアクセスのみ、コンシューマにはみりアクセスのみがあります

クラスとインタフェースの

ジェネリックスをするコードは、ジェネリックコードよりもくがあります。はなです

コンパイルのチェックの

Javaコンパイラは、ジェネリックコードになチェックをし、コードがのにするとエラーをします。コンパイルのエラーをすることは、のエラーをするよりもです。つけにくいがあります。

キャストの

ジェネリックスをしないのコードスニペットでは、キャストがです。

```

List list = new ArrayList();
list.add("hello");

```

```
String s = (String) list.get(0);
```

ジェネリックをするようにすると、コードにはキャストはありません。

```
List<String> list = new ArrayList<>();  
list.add("hello");  
String s = list.get(0);    // no cast
```

プログラマーがアルゴリズムをできるようにする

ジェネリックをすることにより、プログラマーは、さまざまなタイプのコレクションでし、カスタマイズでき、タイプセーフでみやすくなりやすいアルゴリズムをできます。

1つのジェネリックパラメータをバインドする

パラメータは、`T extends Type1 & Type2 & ...`をして、のにバインドすることもできます。

`Flushable`と`Closeable`をするGenericタイプのクラスをしたいと思います。

```
class ExampleClass<T extends Flushable & Closeable> {  
}
```

これで、`ExampleClass`は、`Flushable`と`Closeable`をするタイプをなパラメータとしてのみけれます。

```
ExampleClass<BufferedWriter> arg1; // Works because BufferedWriter implements both Flushable  
and Closeable
```

```
ExampleClass<Console> arg4; // Does NOT work because Console only implements Flushable  
ExampleClass<ZipFile> arg5; // Does NOT work because ZipFile only implements Closeable
```

```
ExampleClass<Flushable> arg2; // Does NOT work because Closeable bound is not satisfied.  
ExampleClass<Closeable> arg3; // Does NOT work because Flushable bound is not satisfied.
```

クラスメソッドは、ジェネリックのを`Closeable`または`Flushable`としてすることができます。

```
class ExampleClass<T extends Flushable & Closeable> {  
    /* Assign it to a valid type as you want. */  
    public void test (T param) {  
        Flushable arg1 = param; // Works  
        Closeable arg2 = param; // Works too.  
    }  
  
    /* You can even invoke the methods of any valid type directly. */  
    public void test2 (T param) {
```

```

        param.flush(); // Method of Flushable called on T and works fine.
        param.close(); // Method of Closeable called on T and works fine too.
    }
}

```

OR | をして、パラメータをいずれかのにバインドすることはできません。AND & のみがサポートされています。は、1つのクラスと多くのインタフェースだけをできます。クラスはリストのになければなりません。

ジェネリックのインスタンス

タイプのため、はしません。

```

public <T> void genericMethod() {
    T t = new T(); // Can not instantiate the type T.
}

```

タイプ_Tはされます。に、JVMは_Tの_Tがであったかをらないので、どのコンストラクタをびすべきかからない。

1. genericMethodをびすときに_TのクラスをgenericMethod

```

public <T> void genericMethod(Class<T> cls) {
    try {
        T t = cls.newInstance();
    } catch (InstantiationException | IllegalAccessException e) {
        System.err.println("Could not instantiate: " + cls.getName());
    }
}

```

```
genericMethod(String.class);
```

されたクラスにアクセスなデフォルトコンストラクタがあるかどうかをるがないため、をスローします。

Java SE 8

2. _Tのコンストラクタへのをす

```

public <T> void genericMethod(Supplier<T> cons) {
    T t = cons.get();
}

```

```
genericMethod(String::new);
```

それのでされたジェネリックをする

メソッドのでされたジェネリックのインスタンスを、されているジェネリックにどのようにって

いますかこれはジェネリックスにしくりげたときにするの1つですが、かなりです。

ここでは、タイプ T をむデータをする`DataSet<T>`ここではインタフェースがあるとしします。

`double`のでくのをするは、このでするのはです。したがって、`DoubleSeries` extends `DataSet<Double>`をします。の`DataSet<T>`には、じさののをしてしいをす `add(values)` というメソッドがあるとしします。どのようにクラスの`DataSet<Double>`ではなく、`values`のとり`values`のを`DoubleSeries`するか

このは、されているをしてするジェネリックパラメータをすることでできますここではインターフェイスにされますが、じものはクラスをします。

```
public interface DataSet<T, DS extends DataSet<T, DS>> {
    DS add(DS values);
    List<T> data();
}
```

ここで、 T はがするデータをし、えは`Double`や`DS`はそのものをす。のパラメータをすることでえることにより、された1つまたはのをにできるようになり、な`Double`ベースのののがられます。

```
public interface DoubleSeries extends DataSet<Double, DoubleSeries> {
    static DoubleSeries instance(Collection<Double> data) {
        return new DoubleSeriesImpl(data);
    }
}
```

では、IDEでもしいタイプのなインターフェイスをしています。ちょっとしたコンテンツのめみには、のようになります。

```
class DoubleSeriesImpl implements DoubleSeries {
    private final List<Double> data;

    DoubleSeriesImpl(Collection<Double> data) {
        this.data = new ArrayList<>(data);
    }

    @Override
    public DoubleSeries add(DoubleSeries values) {
        List<Double> incoming = values != null ? values.data() : null;
        if (incoming == null || incoming.size() != data.size()) {
            throw new IllegalArgumentException("bad series");
        }
        List<Double> newdata = new ArrayList<>(data.size());
        for (int i = 0; i < data.size(); i++) {
            newdata.add(this.data.get(i) + incoming.get(i)); // beware autoboxing
        }
        return DoubleSeries.instance(newdata);
    }

    @Override
    public List<Double> data() {
        return Collections.unmodifiableList(data);
    }
}
```

このように、`add`メソッドは`DoubleSeries add(DoubleSeries values)`としてされており、コンパイラはしています。

にじてパターンをさらにネストすることができます。

Genericsでのinstanceofの

ジェネリックをして**instanceof**でをする

パラメータ`<T>`されたのクラスの`Example`えてみましょう。

```
class Example<T> {
    public boolean isTypeAString(String s) {
        return s instanceof T; // Compilation error, cannot use T as class type here
    }
}
```

コンパイラがJavaソースをJavaバイトコードにコンパイルするとすぐに、タイプと呼ばれるプロセスがされ、すべてのジェネリックコードがジェネリックコードにされ、に`T`タイプをすることがになるため、コンパイルエラーがします。`instanceof`されるは、**reifiable**でなければなりません。**つまり**、にするすべてののをにできるようにするがあります。、これはジェネリックではありません。

のクラスは、つのなるクラスのものをす`Example`、`Example<String>`と`Example<Number>`、ジェネリックは、によりした、のようになります。

```
class Example { // formal parameter is gone
    public boolean isTypeAString(String s) {
        return s instanceof Object; // Both <String> and <Number> are now Object
    }
}
```

タイプがなくなったので、JVMがどのタイプが`T`かをすることはできません。

のルールの

のように、`instanceof`のをするために、のワイルドカードをいつでもできます。

```
public boolean isAList(Object obj) {
    return obj instanceof List<?>;
}
```

これは、インスタンス`obj`が`List`かどうかをするのにです

```
System.out.println(isAList("foo")); // prints false
System.out.println(isAList(new ArrayList<String>())); // prints true
System.out.println(isAList(new ArrayList<Float>())); // prints true
```

、のワイルドカードは、なとみなされます。

instanceofでインスタンスをする

のは、インスタンスすることである`t`の`T`する`instanceof`、のにすように、です。

```
class Example<T> {
    public boolean isTypeAString(T t) {
        return t instanceof String; // No compilation error this time
    }
}
```

タイプののにクラスはのようになります

```
class Example { // formal parameter is gone
    public boolean isTypeAString(Object t) {
        return t instanceof String; // No compilation error this time
    }
}
```

たとえのがこっても、JVMはじ `Object` をしていても、のスニペットにすように、メモリのなるをできるようにになりました。

```
Object obj1 = new String("foo"); // reference type Object, object type String
Object obj2 = new Integer(11); // reference type Object, object type Integer
System.out.println(obj1 instanceof String); // true
System.out.println(obj2 instanceof String); // false, it's an Integer, not a String
```

インターフェースをするためのさまざまなまたはクラスの

のインターフェースがされているとします。

```
public interface MyGenericInterface<T> {
    public void foo(T t);
}
```

に、そのをします。

のをつジェネリッククラスの

ののように、`MyGenericClass`のパラメータ`<T>`をきえるのをしてします。

```
public class NonGenericClass implements MyGenericInterface<String> {
    public void foo(String t) { } // type T has been replaced by String
}
```

このクラスは`String`のみをうので、`MyGenericInterface`をさまざまなパラメータ `Integer`、`Object` などですると、のスニペットがすようにコンパイルされません。

```
NonGenericClass myClass = new NonGenericClass();
myClass.foo("foo_string"); // OK, legal
myClass.foo(11); // NOT OK, does not compile
myClass.foo(new Object()); // NOT OK, does not compile
```

クラスの

のように、MyGenericInterface をするパラメータ<T>をして、のインタフェースをします。

```
public class MyGenericSubclass<T> implements MyGenericInterface<T> {
    public void foo(T t) { } // type T is still the same
    // other methods...
}
```

のように、なるパラメータがされているがあることにしてください。

```
public class MyGenericSubclass<U> implements MyGenericInterface<U> { // equivalent to the
previous declaration
    public void foo(U t) { }
    // other methods...
}
```

ののクラスの

MyGenericInteface ように、MyGenericInteface をするジェネリッククラスをのとしてしますgeneric をまったくしません。

```
public class MyGenericSubclass implements MyGenericInterface {
    public void foo(Object t) { } // type T has been replaced by Object
    // other possible methods
}
```

これは、サブクラスのを インターフェイスの ジェネリックとせてさせるため、に100ではないため、おめできません。のJavaコンパイラはこののをしますが、いJVM1.4とののためにコードはコンパイルされます。

インターフェイスのわりにスーパークラスとしてクラスをするは、のすべてのをすることもできます。

ジェネリックスをしてキャスト

ジェネリックでは、びしがするものをすべてすことができます。

```
private Map<String, Object> data;
public <T> T get(String key) {
    return (T) data.get(key);
}
```

メソッドはとともにコンパイルされます。Javaランタイムは、にキャストをうため、コードはに
はです。

```
Bar bar = foo.get("bar");
```

ジェネリックをするとがします。

```
List<Bar> bars = foo.get("bars");
```

ここで、されるがListいずれかのであるつまり、List<String>をすことでClassCastExceptionトリガ
ーされない、キャストはしClassCastException。

このをするには、きキーをするAPIをします。

```
public final static Key<List<Bar>> BARS = new Key<>("BARS");
```

このput()メソッドとに

```
public <T> T put(Key<T> key, T value);
```

このでは、ったをマップにれることができないので、はにしいでしょうじでなるの2つのキーをつ
てしないり。

- [セーフなマップ](#)

にパラメータをたすクラスをする

メソッドでされているようなくのバインドされていないパラメータは、にはリカバリできません
「のそののスレッド」を。ただし、にクラスのパラメータをたすにアクセスするためののがあり
ます。これにより、すべてのびしをじてをスレッドすることなく、へのアクセスにするコードが
になります。

バックグラウンド

クラスのなパラメータは、のクラスをすることによってできます。このクラスはをします。に、
このメカニズムはスーパータイプのトークンとばれ、[Neal Gafterのブログ](#)にしくされています。

Javaでの3つのなはのとおりです。

- [グアバのタイプトークン](#)
- [SpringのParameterizedTypeReference](#)
- [ジャクソンのタイプ](#)

```
public class DataService<MODEL_TYPE> {  
    private final DataDao dataDao = new DataDao();  
    private final Class<MODEL_TYPE> type = (Class<MODEL_TYPE>) new TypeToken<MODEL_TYPE>  
        (getClass()) {}.getRawType();  
}
```

```
    public List<MODEL_TYPE> getAll() {
        return dataDao.getAllOfType(type);
    }
}

// the subclass definitively binds the parameterization to User
// for all instances of this class, so that information can be
// recovered at runtime
public class UserService extends DataService<User> {}

public class Main {
    public static void main(String[] args) {
        UserService service = new UserService();
        List<User> users = service.getAll();
    }
}
```

オンラインでジェネリックスをむ <https://riptutorial.com/ja/java/topic/92/ジェネリックス>

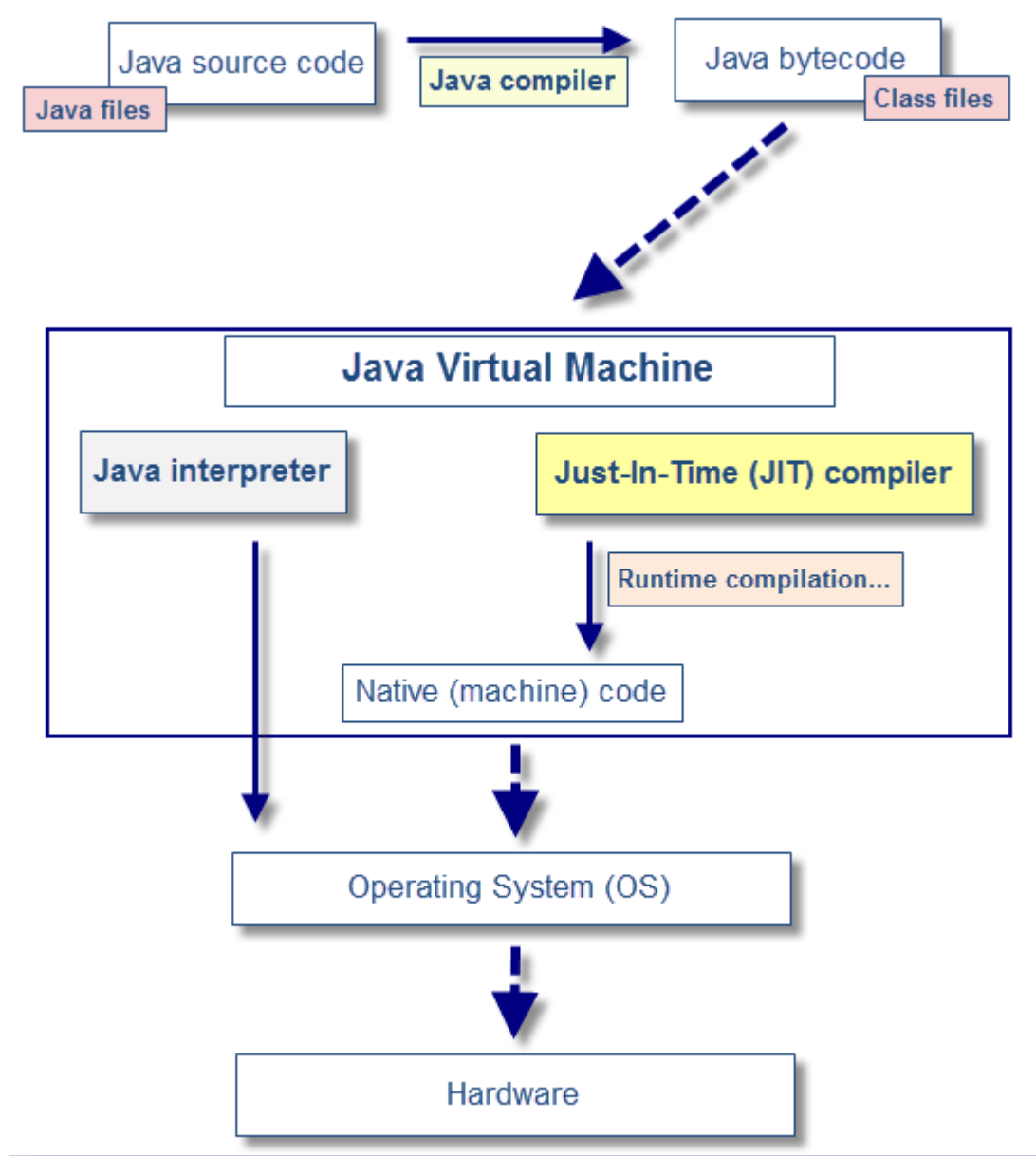
97: ジャストインタイムJITコンパイラ

Symantec JITコンパイラはSun Java 1.1.5でできましたが、がありました。

Hotspot JITコンパイラはプラグインとしてSun Java 1.2.2にされました。Java 1.3では、JITはデフォルトでになっていました。

JavaはいつJITコンパイラをしましたか

Examples



Just-In-TimeJITコンパイラは、にJavaアプリケーションのパフォーマンスをさせるJava™Runtime Environmentのコンポーネントです。

- Javaプログラムは、さまざまなコンピュータアーキテクチャのJVMによってされるプラットフォーム

トフォームバイトコードをむクラスでされています。

- に、JVMはクラスファイルをロードし、々のバイトコードのセマンティクスをし、なをします。

ののプロセッサとメモリーは、Javaアプリケーションがネイティブ・アプリケーションよりもゆっくりとされることをします。

JITコンパイラは、にバイトコードをネイティブのマシンコードにコンパイルすることにより、Javaプログラムのパフォーマンスをさせます。

JITコンパイラはデフォルトでになっており、Javaメソッドがびされるとアクティブになります。JITコンパイラは、そのメソッドのバイトコードをネイティブマシンコードにコンパイルし、するために"just in time"コンパイルします。

メソッドがコンパイルされると、JVMはそれをするのではなく、そのメソッドのコンパイルみコードをびします。には、コンパイルにプロセッサとメモリーがないは、すべてのメソッドをコンパイルするとJavaプログラムのがネイティブ・アプリケーションのにつくがあります。

JITコンパイルにはプロセッサとメモリーがです。JVMがにすると、ものメソッドがびされます。これらのメソッドをすべてコンパイルすると、たとえプログラムがににれたピークをしたとしても、にきなをえるがあります。

-
- には、メソッドはにびされたときにはコンパイルされません。メソッドごとに、JVMはメソッドがびされるたびにインクリメントされるcall countをしcall count。
 - コール・カウントがJITコンパイルのしきいをえるまで、JVMはメソッドをします。
 - したがって、にされるメソッドは、JVMのにコンパイルされ、あまりされないメソッドは、でコンパイルされるか、まったくコンパイルされません。
 - JITコンパイルのしきいは、JVMのをくし、パフォーマンスをさせるのにちます。
 - ととのなバランスをるために、しきいがにされています。
 - メソッドがコンパイルされると、そのびしカウントはゼロにリセットされ、メソッドへのそののびしはそのカウントをしけます。
 - メソッドのコール・カウントがJITコンパイル・スレッシュヨルドにすると、JITコンパイラはそれを2コンパイルして、のコンパイルよりもくのをします。
 - このプロセスは、のレベルにするまでりされる。

Javaプログラムのものいメソッドは、にもにされ、JITコンパイラをすることによるパフォーマンスのをにします。

JITコンパイラはoperational data at run timeoperational data at run time、そのデータをして、コンパイルのをさせることもできます。

JITコンパイラをにすることができます。この、Javaプログラムがされます。JITコンパイルのをまたはするをいて、JITコンパイラをにすることはおめしません。

オンラインでジャストインタイムJITコンパイラをむ <https://riptutorial.com/ja/java/topic/5152/ジャ>

98: シリアライゼーション

き

Javaは、オブジェクトのとばれるメカニズムをします。このメカニズムでは、オブジェクトは、オブジェクトのデータと、オブジェクトのタイプおよびオブジェクトにされているデータのタイプにするをむのバイトとしてできます。

シリアライズされたオブジェクトがファイルにきまれた、ファイルからみみ、デシリアライズすることができます。つまり、オブジェクトをすタイプとバイトとそのデータをしてメモリのオブジェクトをできます。

Examples

Javaでのなシリアライゼーション

シリアライゼーションとは

シリアライゼーションは、オブジェクトのをむのをバイトにするプロセスであり、のあるでこれらのバイトをライブオブジェクトにするプロセスです。シリアルは、オブジェクトをするにされます。Java RMIは、クライアントからサーバへのメソッドびしとして、またはメソッドびしからのりとして、またはリモートメソッドによってスローされるとして、JVMでオブジェクトをすためにもされます。に、JVMのをえてオブジェクトがするようにするには、がされます。

`java.io.Serializable`はマーカーインターフェイスではありません。これは、Javaクラスをシリアライズとしてマークするためにされます。

シリアライズクラスでシリアライズランタイムけバージョンは、びさ `serialVersionUID` シリアライズされたオブジェクトのとのシリアルにしてがあり、そのオブジェクトのクラスをロードしたことをするために、デ-serializationにされます、。が、するのクラスとなる `serialVersionUID` をつオブジェクトのクラスをロードした、により `InvalidClassException` がします。クラスは、のできる `serialVersionUID` というフィールドによってに `serialVersionUID` でなければならない `static, final,` およびタイプの `long`

```
ANY-ACCESS-MODIFIER static final long serialVersionUID = 1L;
```

クラスをににする

オブジェクトをするには、それぞれのクラスが `java.io.Serializable` インタフェースをするがあります。

```
import java.io.Serializable;

public class SerialClass implements Serializable {
```

```

private static final long serialVersionUID = 1L;
private Date currentTime;

public SerialClass() {
    currentTime = Calendar.getInstance().getTime();
}

public Date getCurrentTime() {
    return currentTime;
}
}

```

オブジェクトをファイルにきむ

このオブジェクトをファイルシステムにきむがあります。こので `java.io.ObjectOutputStream` をします。

```

import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.IOException;

public class PersistSerialClass {

    public static void main(String [] args) {
        String filename = "time.ser";
        SerialClass time = new SerialClass(); //We will write this object to file system.
        try {
            ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(filename));
            out.writeObject(time); //Write byte stream to file system.
            out.close();
        } catch (IOException ex){
            ex.printStackTrace();
        }
    }
}

```

されたからオブジェクトをする

されたオブジェクトは、で `java.io.ObjectInputStream` をしてファイルシステムからみることができ
ます。

```

import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.IOException;
import java.io.java.lang.ClassNotFoundException;

public class ReadSerialClass {

    public static void main(String [] args) {
        String filename = "time.ser";
        SerialClass time = null;

        try {
            ObjectInputStream in = new ObjectInputStream(new FileInputStream(filename));
            time = (SerialClass)in.readObject();
            in.close();
        }
    }
}

```

```

    } catch(IOException ex){
        ex.printStackTrace();
    } catch(ClassNotFoundException cnfe){
        cnfe.printStackTrace();
    }
    // print out restored time
    System.out.println("Restored time: " + time.getTime());
}
}

```

シリアル化されたクラスはバイナリです。クラスがされた、はになることがあります。しくは、[Java Serialization SpecificationのSerializable Objectsのバージョンの](#)をしてください。

オブジェクトをシリアル化すると、それがルートであるオブジェクトグラフがシリアルされ、グラフのでしくします。 `reset()` メソッドは、すでにシリアルされているオブジェクトについて `ObjectOutputStream` されるようにします。

フィールド -

Gsonによるシリアル化

Gsonによるシリアル化はで、しいJSONをします。

```

public class Employee {

    private String firstName;
    private String lastName;
    private int age;
    private BigDecimal salary;
    private List<String> skills;

    //getters and setters
}

```

シリアル化

```

//Skills
List<String> skills = new LinkedList<String>();
skills.add("leadership");
skills.add("Java Experience");

//Employee
Employee obj = new Employee();
obj.setFirstName("Christian");
obj.setLastName("Lusardi");
obj.setAge(25);
obj.setSalary(new BigDecimal("10000"));
obj.setSkills(skills);

//Serialization process
Gson gson = new Gson();
String json = gson.toJson(obj);
//{"firstName":"Christian","lastName":"Lusardi","age":25,"salary":10000,"skills":["leadership","Java Experience"]}

```

をしてオブジェクトをシリアルすることはできません。がにするためです。

```
//it's very simple...
//Assuming that json is the previous String object....

Employee obj2 = gson.fromJson(json, Employee.class); // obj2 is just like obj
```

Jackson 2のシリアルライズ

は、オブジェクトをするJSONにシリアルライズするをすです。

```
class Test {

    private int idx;
    private String name;

    public int getIdx() {
        return idx;
    }

    public void setIdx(int idx) {
        this.idx = idx;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}
```

シリアルライゼーション

```
Test test = new Test();
test.setIdx(1);
test.setName("abc");

ObjectMapper mapper = new ObjectMapper();

String jsonString;
try {
    jsonString = mapper.writerWithDefaultPrettyPrinter().writeValueAsString(test);
    System.out.println(jsonString);
} catch (JsonProcessingException ex) {
    // Handle Exception
}
```

```
{
  "idx" : 1,
  "name" : "abc"
}
```

あなたがそれをとしないは、デフォルトプリティプリンターをすることができます。

ここでされるはのとおりで。

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.6.3</version>
</dependency>
```

カスタムシリアル

このでは、にとしてされる2つのののをしてコンソールにするクラスをします。

```
public class SimpleRangeRandom implements Runnable {
    private int min;
    private int max;

    private Thread thread;

    public SimpleRangeRandom(int min, int max){
        this.min = min;
        this.max = max;
        thread = new Thread(this);
        thread.start();
    }

    @Override
    private void WriteObject(ObjectOutputStream out) throws IO Exception;
    private void ReadObject(ObjectInputStream in) throws IOException, ClassNotFoundException;
    public void run() {
        while(true) {
            Random rand = new Random();
            System.out.println("Thread: " + thread.getId() + " Random:" + rand.nextInt(max -
min));
            try {
                Thread.sleep(10000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

このクラスをシリアルライズにしたいは、いくつかのがあります。スレッドは、シリアルライズではないのシステムレベルのクラスの1つです。したがって、スレッドをにするがあります。これを行うことで、このクラスのオブジェクトをシリアルライズすることができますが、ききがします。コンストラクタであることができるように、たちのランダムプログラムのをし、そのにランダムをしてするスレッドをします。したがって、**readObject**をびしてオブジェクトをするとき、コンストラクタはしいオブジェクトのがないのでされません。その、クラスに2つのメソッドをしてカスタムシリアルをするがあります。これらのメソッドはのとおりで。

```
private void writeObject(ObjectOutputStream out) throws IOException;
private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException;
```

したがって、たちのを**readObject**にすることで、スレッドをしてすることができます

```
class RangeRandom implements Serializable, Runnable {

    private int min;
    private int max;

    private transient Thread thread;
    //transient should be any field that either cannot be serialized e.g Thread or any field you
    do not want serialized

    public RangeRandom(int min, int max){
        this.min = min;
        this.max = max;
        thread = new Thread(this);
        thread.start();
    }

    @Override
    public void run() {
        while(true) {
            Random rand = new Random();
            System.out.println("Thread: " + thread.getId() + " Random:" + rand.nextInt(max -
min));
            try {
                Thread.sleep(10000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    private void writeObject(ObjectOutputStream oos) throws IOException {
        oos.defaultWriteObject();
    }

    private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
        in.defaultReadObject();
        thread = new Thread(this);
        thread.start();
    }
}
```

たちののなものはのとおりです。

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello");
        RangeRandom rangeRandom = new RangeRandom(1,10);

        FileOutputStream fos = null;
        ObjectOutputStream out = null;
        try
        {
            fos = new FileOutputStream("test");
            out = new ObjectOutputStream(fos);
            out.writeObject(rangeRandom);
            out.close();
        }
    }
}
```

```

catch(IOException ex)
{
    ex.printStackTrace();
}

RangeRandom rangeRandom2 = null;
FileInputStream fis = null;
ObjectInputStream in = null;
try
{
    fis = new FileInputStream("test");
    in = new ObjectInputStream(fis);
    rangeRandom2 = (RangeRandom)in.readObject();
    in.close();
}
catch(IOException ex)
{
    ex.printStackTrace();
}
catch(ClassNotFoundException ex)
{
    ex.printStackTrace();
}

}
}

```

メインをすると、RangeRandomインスタンスごとに2つのスレッドがされていることがわかります。これは、**Thread.start**メソッドがコンストラクタと**readObject**のにあるためです。

バージョンとserialVersionUID

java.io.Serializableインタフェースをしてクラスをシリアライズにすると、コンパイラはlongのserialVersionUIDというstatic finalフィールドをします。クラスがこのフィールドをにしていな、コンパイラはそのようなフィールドを1つし、serialVersionUIDインプリメンテーションののからりてます。このは、クラスのさまざまなにし、Sunによってえられたオブジェクトにいます。しかし、このは、すべてのコンパイラでじであることがされているわけではありません。

このは、シリアライゼーションにするクラスのをチェックするためにされ、されたオブジェクトのデシリアライズにわれま。ランタイムがいることをしserialVersionUID、デシリアライズされたデータからみられ、serialVersionUIDクラスでまったくじです。そうでないは、InvalidClassExceptionスローします。

このフィールドののデフォルトにるのではなく、シリアライズにするすべてのクラスでlongのなfinalフィールドをにしてし、'serialVersionUID'というをにしてすることをくおめしますバージョンをします。'serialVersionUID'のはにで、コンパイラのごとになるがあります。したがって、プロセスのとでなるコンパイラをしたため、じクラスの中でもInvalidClassExceptionすることがあります。

```

public class Example implements Serializable {
    static final long serialVersionUID = 1L /*or some other value*/;
}

```



```
//...  
}
```

serialVersionUIDがじであれば、Java Serializationはなるバージョンのクラスをうことができます。のあるとのないがあります。

のある

- フィールドのされるクラスにストリームにしないフィールドがある、オブジェクトのフィールドはそのタイプのデフォルトにされます。クラスのがな、クラスはreadObjectメソッドをして、フィールドをデフォルトにできます。
- クラスのストリームには、ストリームのオブジェクトのタイプがまれます。ストリームのこののをクラスとすると、のクラスをできます。ストリームにオブジェクトをするがないため、クラスのフィールドはデフォルトにされます。
- クラスのストリームのクラスをのクラスのクラスとすることで、クラスがされたことをできます。この、そのクラスにするフィールドとオブジェクトがストリームからみられます。プリミティブフィールドはされますが、されたクラスによってされるオブジェクトはストリームのでされるためされます。ストリームがガベージコレクションまたはリセットされると、ガベージコレクションされます。
- **writeObject / readObject** メソッドのストリームをみるバージョンにこれらのメソッドがある、のようにreadObjectがデフォルトのシリアルによってストリームにきまれたなデータを見ることがされます。のデータをみむに、defaultReadObjectをにびすがあります。writeObjectメソッドはどおり、defaultWriteObjectをびしてなデータをきんだ、オプションのデータをきむことができます。
- **java.io.Serializable**のこれはのとじです。このクラスのストリームにはがしないため、フィールドはデフォルトにされます。なクラスのサブクラスのサポートでは、クラスのスーパータイプになしのコンストラクタがあり、クラスがデフォルトにされているがあります。なしのコンストラクタができない、InvalidClassExceptionがスローされます。
- フィールドへのアクセスのアクセスpublic、package、protected、およびprivateは、フィールドにをりてるシリアライズにをえません。
- フィールドを**static**から**nonstatic**または**transient**から**nontransient**にするデフォルトのシリアルをしてシリアライズなフィールドをする、このはクラスにフィールドをすることとじです。しいフィールドはストリームにきまれますが、シリアルではフィールドまたはフィールドにがりてられないため、のクラスではがされます。

のない

- フィールドのクラスでフィールドがされた、きまれたストリームにはそのはまれません。のクラスによってストリームがみられると、ストリームにがないため、フィールドのはデフォルトにされます。しかし、このデフォルトは、をするためののバージョンのをなうがあります。
- クラスをのまたはにするストリームのデータがったでされるため、これはされません。
- フィールドをまたはフィールドにするデフォルトのにする、このはクラスからフィールドを

することとします。このバージョンのクラスは、そのデータをストリームにきまないため、クラスのバージョンではみません。フィールドをするときのように、のバージョンのフィールドはデフォルトにされ、しないでクラスがするがあります。

- プリミティブフィールドのされたのクラスのバージョンはされたのデータをきみます。ストリームのデータのがフィールドのとしなため、フィールドをみろうとするクラスのバージョンはします。
- `writeObject`メソッドまたは`readObject`メソッドをして、のフィールドデータのきみまたはみりをわないようにするか、のバージョンのデータをきもうとしないようにします。デフォルトのフィールドデータは、してストリームにされるか、またはされないようにするがあります。
- クラスには、なクラスのとのないデータがまれるため、クラスを`Serializable`から`Externalizable`に、またはそのにすることはのないです。
- ストリームにはなクラスのとのないデータがまれるため、クラスを`enum`から`enum`に、またはそのにします。
- `Serializable`または`Externalizable`のいずれかをすると、のないになります。これは、きまれると、いバージョンのクラスでなフィールドをしなくなるためです。
- `writeReplace`メソッドまたは`readResolve`メソッドをクラスにすると、クラスのいバージョンとのないオブジェクトがされる、がありません。

JacksonとのカスタムJSONシリアル

たちはJSONとしてAPIをいり、それをPOJOにします。ジャクソンの`org.codehaus.jackson.map.ObjectMapper`は "すぐに"し、ほとんどのにはもしません。しかし、たちのカスタムニーズをたすためにカスタムデシリアライザがなもあります。このチュートリアルでは、のカスタムデシリアライザをするプロセスについてします。

たちにのようなエンティティがあるとしましょう。

```
public class User {
    private Long id;
    private String name;
    private String email;

    //getter setter are omitted for clarity
}
```

そして

```
public class Program {
    private Long id;
    private String name;
    private User createdBy;
    private String contents;

    //getter setter are omitted for clarity
}
```

にオブジェクトを/しましょう。

```

User user = new User();
user.setId(1L);
user.setEmail("example@example.com");
user.setName("Bazlur Rahman");

Program program = new Program();
program.setId(1L);
program.setName("Program @# 1");
program.setCreatedBy(user);
program.setContents("Some contents");

ObjectMapper objectMapper = new ObjectMapper();

```

なString json = objectMapper.writeValueAsStringプログラム; System.out.printlnjson;

のコードは、のJSON-

```

{
  "id": 1,
  "name": "Program @# 1",
  "createdBy": {
    "id": 1,
    "name": "Bazlur Rahman",
    "email": "example@example.com"
  },
  "contents": "Some contents"
}

```

すぐのことをににうことができます。このJSONがある、ObjectMapperをしてのようにプログラムオブジェクトにできます。

これはのケースではないとしましょう。 ProgramクラスとしないAPIとはなるJSONをします。

```

{
  "id": 1,
  "name": "Program @# 1",
  "ownerId": 1
  "contents": "Some contents"
}

```

JSONをと、ownerIdというのフィールドがあります。

はこのJSONをシリアルしたいは、があります。

をし、これをシリアルするには2つのがあります。

のフィールドをする

ownerIdしてownerId。 Programクラスにのアノテーションをする

```

@JsonIgnoreProperties(ignoreUnknown = true)
public class Program {}

```

カスタムデシリアライザの

しかし、にこの`ownerId`フィールドがながあります。これを`User`クラスのIDとしてづけたいとしましょう。

そのようなは、カスタムデシリアライザをするがあります。

このとおり、まず`JsonNode`から`JsonNode`にアクセスする`JsonNode`があり`JsonParser`。そして、`get()`メソッドを使って`JsonNode`からをにすることができます。フィールドをするがあります。それはなでなければなりません、スペルミスがをきこします。

に、あなたの`ProgramDeserializer`を`ObjectMapper`にするがあります。

```
ObjectMapper mapper = new ObjectMapper();
SimpleModule module = new SimpleModule();
module.addDeserializer(Program.class, new ProgramDeserializer());

mapper.registerModule(module);

String newJsonString = "{\"id\":1,\"name\":\"Program @# 1\",\"ownerId\":1,\"contents\":\"Some contents\"}";
final Program program2 = mapper.readValue(newJsonString, Program.class);
```

また、をしてデシリアライザをすることもできます。

```
@JsonDeserialize(using = ProgramDeserializer.class)
public class Program {
}
```

オンラインでシリアライゼーションをむ <https://riptutorial.com/ja/java/topic/767/シリアライゼーション>

99: シングルトン

き

シングルトンは、1つのインスタンスのみをつクラスです。シングルトンデザインパターンについては、「[デザインパターン](#)」タグの「[シングルトン](#)」トピックをしてください。

Examples

シングルトン

Java SE 5

```
public enum Singleton {  
    INSTANCE;  
  
    public void execute (String arg) {  
        // Perform operation here  
    }  
}
```

はプライベートコンストラクタをち、なものであり、なシリアルをします。それらはにであり、スレッドセーフなでにされます。

JVMは、のがそれぞれインスタンスされないことをします。これにより、のシングルトンパターンはにしてになをします。

enumパターンがしないものは、にソースコードにをしているのです。そのため、シングルトンにこのスタイルをした、これらののにしいをしないことをにすることがです。

これは、Joshua BlochがEffective Javaでしたように、シングルトンパターンをするためのされるです。

ダブルチェックロックきスレッドセーフシングルトン

このタイプのシングルトンはスレッドセーフであり、シングルトンインスタンスがされたのなロックをぎます。

Java SE 5

```
public class MySingleton {  
  
    // instance of class  
    private static volatile MySingleton instance = null;  
  
    // Private constructor  
    private MySingleton() {
```

```

        // Some code for constructing object
    }

    public static MySingleton getInstance() {
        MySingleton result = instance;

        //If the instance already exists, no locking is necessary
        if(result == null) {
            //The singleton instance doesn't exist, lock and check again
            synchronized(MySingleton.class) {
                result = instance;
                if(result == null) {
                    instance = result = new MySingleton();
                }
            }
        }
        return result;
    }
}

```

しなければならないのは、Java SE 5よりのバージョンでは、のがしくないため、けるべきです。Java 5よりのJavaでは、ダブルチェックロックをしすることはできません。

をしないシングルトンな

```

public class Singleton {

    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}

```

このはであるとできます。 [Javaの12.4.1](#)はのようにべています。

クラスまたはインタフェースタイプTは、のいずれかがにれるにされます。

- Tはクラスであり、Tのインスタンスがされる
- Tはクラスであり、Tによってされたメソッドがびされる
- Tでされたフィールドがりてられます。
- Tでされたフィールドがされ、フィールドはではありません
- Tはクラスであり、Tににネストされたアサーションがされる。

したがって、クラスにのフィールドやメソッドがないり、めて `getInstance()` メソッドがびされるまで、Singletonインスタンスはされません。

ホルダークラスをしたスレッドセーフな| Bill Pughシングルトンの

```

public class Singleton {

```

```

private static class InstanceHolder {
    static final Singleton INSTANCE = new Singleton();
}

public static Singleton getInstance() {
    return InstanceHolder.INSTANCE;
}

private Singleton() {}
}

```

これにより、`Singleton.getInstance()` への呼び出しで `INSTANCE` がされ、のをとせずににするのスレッドがされます。

これはBill Pughシングルトンパターンともされます。 [\[ウィキ\]](#)

シングルトンのシングルトン

ここでは、クラスの `Singleton` は "Hello world!" を `getMessage()` メソッドをしています "Hello world!" メッセージ。

サブクラス `UppercaseSingleton` と `LowercaseSingleton` は、メッセージのなをする `getMessage` メソッドをオーバーライドします。

```

//Yeah, we'll need reflection to pull this off.
import java.lang.reflect.*;

/*
 * Enumeration that represents possible classes of singleton instance.
 * If unknown, we'll go with base class - Singleton.
 */
enum SingletonKind {
    UNKNOWN,
    LOWERCASE,
    UPPERCASE
}

//Base class
class Singleton{

    /*
     * Extended classes has to be private inner classes, to prevent extending them in
     * uncontrolled manner.
     */
    private class UppercaseSingleton extends Singleton {

        private UppercaseSingleton(){
            super();
        }

        @Override
        public String getMessage() {
            return super.getMessage().toUpperCase();
        }
    }
}

```

```
//Another extended class.
private class LowercaseSingleton extends Singleton
{
    private LowercaseSingleton(){
        super();
    }

    @Override
    public String getMessage() {
        return super.getMessage().toLowerCase();
    }
}

//Applying Singleton pattern
private static SingletonKind kind = SingletonKind.UNKNOWN;

private static Singleton instance;

/*
By using this method prior to getInstance() method, you effectively change the
type of singleton instance to be created.
*/
public static void setKind(SingletonKind kind) {
    Singleton.kind = kind;
}

/*
If needed, getInstance() creates instance appropriate class, based on value of
singletonKind field.
*/
public static Singleton getInstance()
    throws  NoSuchMethodException,
           IllegalAccessException,
           InvocationTargetException,
           InstantiationException {

    if(instance==null){
        synchronized (Singleton.class){
            if(instance==null){
                Singleton singleton = new Singleton();
                switch (kind){
                    case UNKNOWN:

                        instance = singleton;
                        break;

                    case LOWERCASE:

                        /*
                        I can't use simple

                        instance = new LowercaseSingleton();

                        because java compiler won't allow me to use
                        constructor of inner class in static context,
                        so I use reflection API instead.

                        To be able to access inner class by reflection API,
                        I have to create instance of outer class first.
                        Therefore, in this implementation, Singleton cannot be
                        abstract class.

```



```

        */

        //Get the constructor of inner class.
        Constructor<LowercaseSingleton> lcConstructor =
LowercaseSingleton.class.getDeclaredConstructor(Singleton.class);

        //The constructor is private, so I have to make it accessible.
        lcConstructor.setAccessible(true);

        // Use the constructor to create instance.
        instance = lcConstructor.newInstance.singleton);

        break;

    case UPPERCASE:

        //Same goes here, just with different type
        Constructor<UppercaseSingleton> ucConstructor =
UppercaseSingleton.class.getDeclaredConstructor(Singleton.class);
        ucConstructor.setAccessible(true);
        instance = ucConstructor.newInstance.singleton);
    }
}
}
return instance;
}

//Singletons state that is to be used by subclasses
protected String message;

//Private constructor prevents external instantiation.
private Singleton()
{
    message = "Hello world!";
}

//Singleton's API. Implementation can be overwritten by subclasses.
public String getMessage() {
    return message;
}
}

//Just a small test program
public class ExtendingSingletonExample {

    public static void main(String args[]){

        //just uncomment one of following lines to change singleton class

        //Singleton.setKind(SingletonKind.UPPERCASE);
        //Singleton.setKind(SingletonKind.LOWERCASE);

        Singleton singleton = null;
        try {
            singleton = Singleton.getInstance();
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {

```

```
        e.printStackTrace();
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    } catch (InstantiationException e) {
        e.printStackTrace();
    }
    System.out.println(singleton.getMessage());
}
}
```

オンラインでシングルトンをむ <https://riptutorial.com/ja/java/topic/130/シングルトン>

100: スーパーキーワード

Examples

スーパーキーワードの

スーパーキーワードは3つのでなをたす

1. コンストラクタレベル
2. メソッドレベル
3. レベル

コンストラクタレベル

`super` キーワードは、クラスのコンストラクタをびすためにされます。このコンストラクタは、デフォルトコンストラクタまたはパラメータコンストラクタにすることができます。

- デフォルトのコンストラクタ `super();`
- パラメータされたコンストラクタ `super(int no, double amount, String name);`

```
class Parentclass
{
    Parentclass(){
        System.out.println("Constructor of Superclass");
    }
}
class Subclass extends Parentclass
{
    Subclass(){
        /* Compile adds super() here at the first line
        * of this constructor implicitly
        */
        System.out.println("Constructor of Subclass");
    }
    Subclass(int n1){
        /* Compile adds super() here at the first line
        * of this constructor implicitly
        */
        System.out.println("Constructor with arg");
    }
    void display(){
        System.out.println("Hello");
    }
    public static void main(String args[]){
        // Creating object using default constructor
        Subclass obj= new Subclass();
        //Calling sub class method
        obj.display();
        //Creating object 2 using arg constructor
        Subclass obj2= new Subclass(10);
    }
}
```

```

        obj2.display();
    }
}

```

`super()` はコンストラクターのステートメントでなければなりません。さもなければ、コンパイルエラーメッセージがされます。

メソッドレベル

`super` キーワードは、メソッドオーバーライドのにもできます。 `super` キーワードをして、クラス
のメソッドをひすか、ひすことができます。

```

class Parentclass
{
    //Overridden method
    void display(){
        System.out.println("Parent class method");
    }
}
class Subclass extends Parentclass
{
    //Overriding method
    void display(){
        System.out.println("Child class method");
    }
    void printMsg(){
        //This would call Overriding method
        display();
        //This would call Overridden method
        super.display();
    }
    public static void main(String args[]){
        Subclass obj= new Subclass();
        obj.printMsg();
    }
}

```

メソッドのオーバーライドがない、クラスメソッドをひすために `super` キーワードをするはありません。

レベル

`super` は、のクラスのインスタンスをするためにされます。の、クラスがあり、クラスはのデータメンバーをつがあります。1クラスと1クラスのデータメンバーをするために、クラスのコンテキストでは、クラスデータメンバーのには `super` キーワードがです。

```

//Parent class or Superclass
class Parentclass
{
    int num=100;
}
//Child class or subclass

```

```
class Subclass extends Parentclass
{
    /* I am declaring the same variable
    * num in child class too.
    */
    int num=110;
    void printNumber(){
        System.out.println(num); //It will print value 110
        System.out.println(super.num); //It will print value 100
    }
    public static void main(String args[]){
        Subclass obj= new Subclass();
        obj.printNumber();
    }
}
```

クラスのデータメンバのに `super` キーワードをしていない、それはのクラスデータメンバとばれ、ベースクラスのデータメンバはクラスのコンテキストにされます。

オンラインでスーパーキーワードをむ <https://riptutorial.com/ja/java/topic/5764/スーパーキーワード>

101: スキャナ

- スキャナスキャナ=しいスキャナソースソース;
- スキャナスキャナ=しいスキャナSystem.in;

パラメーター

パラメータ	
ソース	Sourceは、String、FileまたはのInputStreamのいずれかです

ScannerクラスはJava 5でされました。Java 6ではreset()メソッドがされ、しいPathインターフェイスとののために、Java 7にはいくつかのしいコンストラクタがされました。

Examples

Scannerをしたシステムののみり

```
Scanner scanner = new Scanner(System.in); //Scanner obj to read System input
String inputTaken = new String();
while (true) {
    String input = scanner.nextLine(); // reading one line of input
    if (input.matches("\\s+"))          // if it matches spaces/tabs, stop reading
        break;
    inputTaken += input + " ";
}
System.out.println(inputTaken);
```

スキャナオブジェクトは、キーボードからをみるようにされます。 keyboardからののにはは Reading from keyboardとしてをします

```
Reading
from
keyboard
//space
```

Scannerをしたファイルのみり

```
Scanner scanner = null;
try {
    scanner = new Scanner(new File("Names.txt"));
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
} catch (Exception e) {
    System.err.println("Exception occurred!");
}
```

```

} finally {
    if (scanner != null)
        scanner.close();
}

```

ここで、`Scanner`オブジェクトは、テキストファイルのをむ`File`オブジェクトをとってすることによってされます。このテキストファイルは`File`オブジェクトによってかれ、スキャナオブジェクトによってのにみまれます。`scanner.hasNext()`は、テキストファイルにのデータがあるかどうかをチェックします。これを`while`ループとみわせると、`Names.txt`ファイルのすべてのデータをりし`Names.txt`できます。データそのものをするには、`nextLine()`、`nextInt()`、`nextBoolean()`などのメソッドをできます。のでは、`scanner.nextLine()`がされています。`nextLine()`は、テキストファイルののをし、それを`scanner`オブジェクトとみわせることで、のをすることができます。スキャナオブジェクトをじるには、`.close()`をします。

Java 7のリソースで`try`をつて、のコードをのようにエレガントにくことができます。

```

try (Scanner scanner = new Scanner(new File("Names.txt"))) {
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
} catch (Exception e) {
    System.err.println("Exception occurred!");
}

```

Scannerをしてをとしてみる

`Scanner`をすると、`\Z`をりとしてして、のすべてのテキストをとしてみることができます。たとえば、これをして、テキストファイルのすべてのテキストを1でみむことができます。

```

String content = new Scanner(new File("filename")).useDelimiter("\\Z").next();
System.out.println(content);

```

Scannerをしてファイルをみむでしているように、`Scanner`をし、`IOException`がスローされるがあることに`IOException`。

カスタムりの

`.useDelimiter(",")`、`Scanner`でカスタムりをして、のみみをできます。これは`String.split(...)`とし`String.split(...)`。たとえば、`Scanner`をして、のコンマりのリストからみむことができます。

```

Scanner scanner = null;
try{
    scanner = new Scanner("i,like,unicorns").useDelimiter(",");
    while(scanner.hasNext()){
        System.out.println(scanner.next());
    }
}catch(Exception e){
    e.printStackTrace();
}

```

```

}finally{
    if (scanner != null)
        scanner.close();
}

```

これにより、のすべてののをにみることができます。あなたは、なCSVパーサライブラリをし、わりに、CSVデータをするために、これをしててはならないことにしてください[JavaのCSVパーサー](#)をののために。

タスクについてもよくねられるなパターン

は、`java.util.Scanner`クラスをにして、`System.in`からのユーザーをにしくみむです `stdin`ともばれます。にCやC++、UnixやLinuxなどのでもされます。これは、がされているもなことをにしています。

```

package com.stackoverflow.scanner;

import javax.annotation.Nonnull;
import java.math.BigInteger;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.*;
import java.util.regex.Pattern;

import static java.lang.String.format;

public class ScannerExample
{
    private static final Set<String> EXIT_COMMANDS;
    private static final Set<String> HELP_COMMANDS;
    private static final Pattern DATE_PATTERN;
    private static final String HELP_MESSAGE;

    static
    {
        final SortedSet<String> ecmds = new TreeSet<String>(String.CASE_INSENSITIVE_ORDER);
        ecmds.addAll(Arrays.asList("exit", "done", "quit", "end", "fino"));
        EXIT_COMMANDS = Collections.unmodifiableSortedSet(ecmds);
        final SortedSet<String> hcmds = new TreeSet<String>(String.CASE_INSENSITIVE_ORDER);
        hcmds.addAll(Arrays.asList("help", "helpi", "?"));
        HELP_COMMANDS = Collections.unmodifiableSet(hcmds);
        DATE_PATTERN = Pattern.compile("\\d{4}([-\\/]\\d{2}\\1\\d{2}"); //
        http://regex101.com/r/xB8dR3/1
        HELP_MESSAGE = format("Please enter some data or enter one of the following commands
to exit %s", EXIT_COMMANDS);
    }

    /**
     * Using exceptions to control execution flow is always bad.
     * That is why this is encapsulated in a method, this is done this
     * way specifically so as not to introduce any external libraries
     * so that this is a completely self contained example.
     * @param s possible url
     * @return true if s represents a valid url, false otherwise
     */
    private static boolean isValidURL(@Nonnull final String s)
    {

```



```

        try { new URL(s); return true; }
        catch (final MalformedURLException e) { return false; }
    }

    private static void output(@Nonnull final String format, @Nonnull final Object... args)
    {
        System.out.println(format(format, args));
    }

    public static void main(final String[] args)
    {
        final Scanner sis = new Scanner(System.in);
        output(HELP_MESSAGE);
        while (sis.hasNext())
        {
            if (sis.hasNextInt())
            {
                final int next = sis.nextInt();
                output("You entered an Integer = %d", next);
            }
            else if (sis.hasNextLong())
            {
                final long next = sis.nextLong();
                output("You entered a Long = %d", next);
            }
            else if (sis.hasNextDouble())
            {
                final double next = sis.nextDouble();
                output("You entered a Double = %f", next);
            }
            else if (sis.hasNext("\\d+"))
            {
                final BigInteger next = sis.nextBigInteger();
                output("You entered a BigInteger = %s", next);
            }
            else if (sis.hasNextBoolean())
            {
                final boolean next = sis.nextBoolean();
                output("You entered a Boolean representation = %s", next);
            }
            else if (sis.hasNext(DATE_PATTERN))
            {
                final String next = sis.next(DATE_PATTERN);
                output("You entered a Date representation = %s", next);
            }
            else // unclassified
            {
                final String next = sis.next();
                if (isValidURL(next))
                {
                    output("You entered a valid URL = %s", next);
                }
                else
                {
                    if (EXIT_COMMANDS.contains(next))
                    {
                        output("Exit command %s issued, exiting!", next);
                        break;
                    }
                    else if (HELP_COMMANDS.contains(next)) { output(HELP_MESSAGE); }
                    else { output("You entered an unclassified String = %s", next); }
                }
            }
        }
    }

```

```

    }
}
}
/*
This will close the underlying Readable, in this case System.in, and free those
resources.
You will not be to read from System.in anymore after this you call .close().
If you wanted to use System.in for something else, then don't close the Scanner.
*/
sis.close();
System.exit(0);
}
}

```

コマンドラインからintをみむ

```

import java.util.Scanner;

Scanner s = new Scanner(System.in);
int number = s.nextInt();

```

コマンドラインからintをみたいは、このスニペットをしてください。まず、コマンドラインからプログラムをすると、デフォルトではコマンドラインであるSystem.inをリスンするScannerオブジェクトをすることがあります。その、Scannerオブジェクトのけをりて、ユーザーがコマンドラインにしたのintをみてにします。これで、されたintをとって、でもできます。

にスキャナをじる

コンストラクタのパラメータとしてSystem.inをしたスキャナをすると、スキャナをじるとInputStreamもじてしまうことにづくがあります。スキャナオブジェクトは、

java.util.NoSuchElementExceptionまたはjava.lang.IllegalStateExceptionをスローし
java.lang.IllegalStateException

```

Scanner sc1 = new Scanner(System.in);
Scanner sc2 = new Scanner(System.in);
int x1 = sc1.nextInt();
sc1.close();
// java.util.NoSuchElementException
int x2 = sc2.nextInt();
// java.lang.IllegalStateException
x2 = sc1.nextInt();

```

オンラインでスキャナをむ <https://riptutorial.com/ja/java/topic/551/スキャナ>

102: ストリーム

き

`Stream`はシーケンスをし、それらののをするためのなるのをサポートします。Java 8では、`Collection`インタフェースに`Stream`をする2つのメソッド `stream()` と `parallelStream()` ます。`Stream`は、またはのいずれかです。は`Stream`すので、`Stream`がじるにのをチェーンすることができます。ターミナルはであるか、ストリームをします。

- `collection.stream`
- 。ストリーム
- `Stream.iterateFirstValue、currentValue -> nextValue`
- `Stream.generate -> value`
- `Stream.ofElementOfT [、elementOfT、...]`
- `Stream.empty`
- `StreamSupport.streamIterable.spliterator、false`

Examples

ストリームの

`Stream`とは、およびのをできるのです。の`Stream`は、データがにれるがあります。その、`Stream`からしたデータは、データのバッチをするのではなく、ににされます。[ラムダ](#)とみわせると、アプローチをしてのデータにするをにうことができます。

[それはIdeoneです](#)をすることをしてください

```
Stream<String> fruitStream = Stream.of("apple", "banana", "pear", "kiwi", "orange");

fruitStream.filter(s -> s.contains("a"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
```

バナナ
オレンジ

のコードでされるは、のようにできます。

1. `Stream<String>`ファクトリメソッド `Stream.of(values)` をして、シーケンスされたけられたフルーツ `String`の`Stream`をむ`Stream<String>`をします。
2. `filter()` オペレーションは、えられたによってテストされたときに`true`をすとするだけをし

ます。この、`"a"`をむはされます。はラムダとしてえられます。

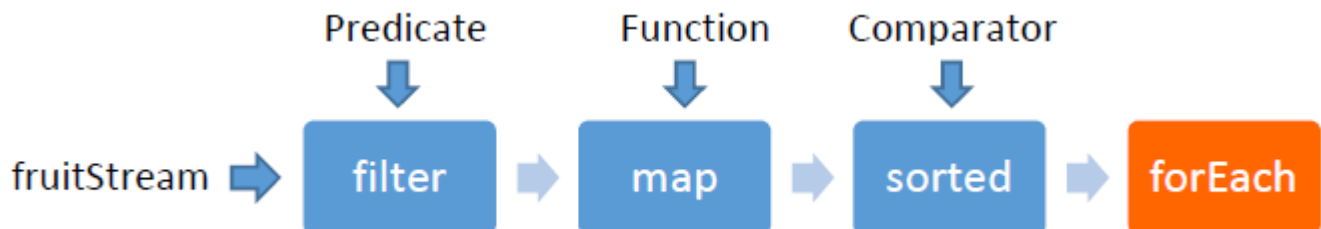
3. `map()` は、マッパーと呼ばれるのをしてをします。この、フルーツ`String`は、メソッド `String::toUpperCase` をして、の`String`バージョンにマップされます。

ことにされたい `map()` マッピングは、パラメータになるタイプをした、はなるなタイプのストリームをします。たとえば、`Stream<String>` で `.map(String::isEmpty)` をすると、`Stream<Boolean>`

4. `sorted()` オペレーションは、け `String`のはに`sorted()` につて、`Stream`のをソートします。

5. に、`forEach(action)` は、`Stream`にしてし、`Consumer`にすアクションをします。このでは、はにコンソールにされています。このはであり、することはできません。

`Stream`されたは、のためにされることにしてください。ターミナルなしでは、ストリームはされません。ストリームはできません。ターミナルがびされると、`Stream`オブジェクトはできなくなります。



はのようにされて、データにするクエリとしてえるものをします。

ストリームをじる

`Stream`にじておくはないことにしてください。IOチャネルでするストリームをじるがあるだけです。ほとんどの`Stream`タイプはリソースでしないため、するはありません。

`Stream`インターフェイスは`AutoCloseable`し`AutoCloseable`。ストリームは、`close`メソッドをびすか、`try-with-resource`ステートメントをして`close`ことができます。

`Stream`をじるがあるユースケースのは、ファイルからの`Stream`をするです。

```
try (Stream<String> lines = Files.lines(Paths.get("somePath"))) {
    lines.forEach(System.out::println);
}
```

`Stream`インターフェイスは、`Stream.onClose()`メソッドもします。このメソッドをすると、ストリームがじられたときにびされる`Runnable`ハンドラをできます。ストリームをするコードがらかのクリーンアップをするためにされたときをるがあるのがあります。

```
public Stream<String>streamAndDelete(Path path) throws IOException {
    return Files.lines(path).onClose(() -> someClass.deletePath(path));
}
```

ハンドラは、try-with-resourcesステートメントによってまたはにclose()メソッドが呼び出されたにのみされます。

の

Streamオブジェクトのは、シーケンシャルまたは**パラレル**でうことができます。

シーケンシャルモードでは、はStreamソースのにされます。StreamがけされているSortedMapやList、はソースのにすることがされます。しかし、のケースでは、にしないようにする**があります**keySet() Java HashMap keySet() **はしていますかをしてください**。

```
List<Integer> integerList = Arrays.asList(0, 1, 2, 3, 42);

// sequential
long howManyOddNumbers = integerList.stream()
    .filter(e -> (e % 2) == 1)
    .count();

System.out.println(howManyOddNumbers); // Output: 2
```

イデオンにきる

パラレルモードでは、のコアでのスレッドをできますが、のをするものではありません。

のメソッドがStreamで呼び出された、すべてのメソッドを呼び出すはありません。たとえば、Streamがフィルタリングされ、のが1にらされた、sortなどのメソッドへののびしはわれません。これにより、StreamパフォーマンスがするがありStream。これは、Streamではなです。

```
// parallel
long howManyOddNumbersParallel = integerList.parallelStream()
    .filter(e -> (e % 2) == 1)
    .count();

System.out.println(howManyOddNumbersParallel); // Output: 2
```

イデオンにきる

コンテナまたはコレクション とのい

いくつかのアクションはコンテナとストリームのでできますが、にはなるをたし、さまざまなをサポートします。コンテナは、がどのようにされているか、それらののにアクセスするにがてられています。、Streamはへのアクセスとをしません。それはエンティティとしてのオブジェクト

のグループにし、そのエンティティにしてをする。 `Stream`と`Collection`は、これらのなるのためのレベルである。

コレクションのをコレクションにめる

`toList()` と `toSet()` する

`Stream`は、 `Stream.collect` をしてコンテナににできます。

```
System.out.println(Arrays
    .asList("apple", "banana", "pear", "kiwi", "orange")
    .stream()
    .filter(s -> s.contains("a"))
    .collect(Collectors.toList())
);
// prints: [apple, banana, pear, orange]
```

`Set`などのコレクションインスタンスは、の`Collectors`みみメソッドをしてできます。たとえば、 `Collectors.toSet()` は`Stream`のを`Set`めます。

`List` または `Set` のな

`Collectors#toList()` と `Collectors#toSet()` ドキュメントによれば、される`List`または`Set`の、、、またはスレッドについてはありません。

されるのなのために、わりにされたサプライヤがしいのコレクションをす `Collectors#toCollection(Supplier)` をできます。

```
// syntax with method reference
System.out.println(strings
    .stream()
    .filter(s -> s != null && s.length() <= 3)
    .collect(Collectors.toCollection(ArrayList::new))
);

// syntax with lambda
System.out.println(strings
    .stream()
    .filter(s -> s != null && s.length() <= 3)
    .collect(Collectors.toCollection(() -> new LinkedHashSet<>()))
);
```

`toMap`をしたの

コレクターはを`Map`にします。ここで、`key`はIDであり、`Value`はです。

```
List<Student> students = new ArrayList<Student>();
students.add(new Student(1, "test1"));
students.add(new Student(2, "test2"));
```

```
students.add(new Student(3, "test3"));

Map<Integer, String> IdToName = students.stream()
    .collect(Collectors.toMap(Student::getId, Student::getName));
System.out.println(IdToName);
```

```
{1=test1, 2=test2, 3=test3}
```

Collector.toMapには `Collector<T, ?, Map<K,U>> toMap(Function<? super T, ? extends K> keyMapper, Function<? super T, ? extends U> valueMapper, BinaryOperator<U> mergeFunction)`。 **mergeFunction** は、リストからマップにしいメンバをするときにキーがりされる、にしいをするかいをするためにされます。

mergeFunctionは、 `(s1, s2) -> s1` ようにりしキーにするをするか、 `(s1, s2) -> s2` りしキーのしいをします。

コレクションのをする

ArrayListから**Map**へ<String、List <>>

しばしば、プライマリリストからリストのマップをするがあります。リストのから、のリストをするがあります。

```
List<Student> list = new ArrayList<>();
list.add(new Student("Davis", SUBJECT.MATH, 35.0));
list.add(new Student("Davis", SUBJECT.SCIENCE, 12.9));
list.add(new Student("Davis", SUBJECT.GEOGRAPHY, 37.0));

list.add(new Student("Sascha", SUBJECT.ENGLISH, 85.0));
list.add(new Student("Sascha", SUBJECT.MATH, 80.0));
list.add(new Student("Sascha", SUBJECT.SCIENCE, 12.0));
list.add(new Student("Sascha", SUBJECT.LITERATURE, 50.0));

list.add(new Student("Robert", SUBJECT.LITERATURE, 12.0));

Map<String, List<SUBJECT>> map = new HashMap<>();
list.stream().forEach(s -> {
    map.computeIfAbsent(s.getName(), x -> new ArrayList<>()).add(s.getSubject());
});
System.out.println(map);
```

```
{ Robert=[LITERATURE],
  Sascha=[ENGLISH, MATH, SCIENCE, LITERATURE],
  Davis=[MATH, SCIENCE, GEOGRAPHY] }
```

ArrayListから**Map** <String、Map <>>

```
List<Student> list = new ArrayList<>();
list.add(new Student("Davis", SUBJECT.MATH, 1, 35.0));
list.add(new Student("Davis", SUBJECT.SCIENCE, 2, 12.9));
list.add(new Student("Davis", SUBJECT.MATH, 3, 37.0));
list.add(new Student("Davis", SUBJECT.SCIENCE, 4, 37.0));
```

```
list.add(new Student("Sascha", SUBJECT.ENGLISH, 5, 85.0));
list.add(new Student("Sascha", SUBJECT.MATH, 1, 80.0));
list.add(new Student("Sascha", SUBJECT.ENGLISH, 6, 12.0));
list.add(new Student("Sascha", SUBJECT.MATH, 3, 50.0));

list.add(new Student("Robert", SUBJECT.ENGLISH, 5, 12.0));

Map<String, Map<SUBJECT, List<Double>>> map = new HashMap<>();

list.stream().forEach(student -> {
    map.computeIfAbsent(student.getName(), s -> new HashMap<>())
        .computeIfAbsent(student.getSubject(), s -> new ArrayList<>())
        .add(student.getMarks());
});

System.out.println(map);
```

```
{ Robert={ENGLISH=[12.0]},
Sascha={MATH=[80.0, 50.0], ENGLISH=[85.0, 12.0]},
Davis={MATH=[35.0, 37.0], SCIENCE=[12.9, 37.0]} }
```

カンニングペーパー

ゴール	コード
Listめる	Collectors.toList()
りてサイズのArrayListする	Collectors.toCollection(() -> new ArrayList<>(size))
Setめる	Collectors.toSet()
よりいでSetめる	Collectors.toCollection(() -> new LinkedHashSet<>())
とをしないSet<String>	Collectors.toCollection(() -> new TreeSet<>(String.CASE_INSENSITIVE_ORDER))
EnumSet<AnEnum>めるでのパフォーマンス	Collectors.toCollection(() -> EnumSet.noneOf(AnEnum.class))
のキーをしてMap<K,V>する	Collectors.toMap(keyFunc, valFunc)
MyObject.getterをのMyObjectにマップする	Collectors.toMap(MyObject::getter, Function.identity())
MyObject.getterをのMyObjectにマップする	Collectors.groupingBy(MyObject::getter)

のれ

しない `Stream` をすることはです。 `Stream` でメソッドをびすと、 `Stream` はループになります。 `Stream` の `limit` メソッドは、Java がする `Stream` ののをするためにできます。

このでは、する `Stream` のターム1からまる、すべてののを `Stream` よりいです。この `Stream limit` メソッドをびすことで、 `Stream` の5つのだけがされ、されます。

```
// Generate infinite stream - 1, 2, 3, 4, 5, 6, 7, ...
IntStream naturalNumbers = IntStream.iterate(1, x -> x + 1);

// Print out only the first 5 terms
naturalNumbers.limit(5).forEach(System.out::println);
```

1
2
3
4
5

ストリームをするのは、 `Stream.generate` メソッドをすることです。このメソッドは、 `Supplier` のラムダをります。

```
// Generate an infinite stream of random numbers
Stream<Double> infiniteRandomNumbers = Stream.generate(Math::random);

// Print out only the first 10 random numbers
infiniteRandomNumbers.limit(10).forEach(System.out::println);
```

ストリームをする

`Stream` は、 `count()` 、 `collect()` 、 `forEach()` などのがあるにのみトラバースされます。それのは、 `Stream` にするはされません。

のでは、ターミナルは `Stream` にされないため、 `peek()` はターミナルではないため、 `filter()` はびされず、もされません。

```
IntStream.range(1, 10).filter(a -> a % 2 == 0).peek(System.out::println);
```

イデオンにきる

これは、なターミナルをつ `Stream` シーケンスであるため、がされます。

`peek` わりに `forEach` をうこともできます

```
IntStream.range(1, 10).filter(a -> a % 2 == 0).forEach(System.out::println);
```

イデオンにきる

2
4
6
8

ターミナルがされた、`Stream`はされ、することはできません。

ストリームオブジェクトをすることはできませんが、ストリームパイプラインにするな`Iterable`をするのはです。これは、なにをすることなく、ライブデータセットのされたビューをすのにです。

```
List<String> list = Arrays.asList("FOO", "BAR");
Iterable<String> iterable = () -> list.stream().map(String::toLowerCase).iterator();

for (String str : iterable) {
    System.out.println(str);
}
for (String str : iterable) {
    System.out.println(str);
}
```

foo
バー
foo
バー

これは、`Iterable`がのメソッド`Iterator<T> iterator()`するために`Iterable`し`Iterator<T> iterator()`。これにより、に、びしでしいストリームをするラムダによってされたなインターフェイスになります。

に、`Stream`はのののようにします。



のチェックは、ターミナルをわなくてもにされます。

```
try {
    IntStream.range(1, 10).filter(null);
} catch (NullPointerException e) {
    System.out.println("We got a NullPointerException as null was passed as an argument to filter()");
}
```

イデオンにきる

filterへのとしてnullがされたため、NullPointerExceptionがしました。

マップの

`groupingBy(classifier, downstream)` コレクタは、グループのをし、じグループにされたにしてをすることによって、`Stream`のを`Map`にすることをにする。

こののなは、`Map`をして`Stream`のをカウントすることです。このでは、クラシファイアはにアイデンティティであり、をそのままします。は、`counting()`をしてしいのを`counting()`ます。

```
Stream.of("apple", "orange", "banana", "apple")
    .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()))
    .entrySet()
    .forEach(System.out::println);
```

は、それがコレクタ `Collectors.counting()` です。この`Collectors.counting()`は、`String`のでし、`Long`のをします。 `collect` メソッドびしのは`Map<String, Long>`です。

これにより、のがされます。

```
バナナ=1
オレンジ=1
リンゴ=2
```

パラレルストリーム

する`Stream`をするに、`ParallelStream`と`Sequential Stream`のをしてください。

`Stream`をにするは、これらののいずれかをできます。

```
List<String> data = Arrays.asList("One", "Two", "Three", "Four", "Five");
Stream<String> aParallelStream = data.stream().parallel();
```

または

```
Stream<String> aParallelStream = data.parallelStream();
```

ストリームにしてされたをするには、をびします。

```
aParallelStream.forEach(System.out::println);
```

`Stream`からのな

すべてのがにされるようがされるがありますどちらがよりくそれをうことができます。がでない
は、 `parallelStream` してください。

パフォーマンスへの

ケースネットワークがしているでは、 `Stream S` は、アプリケーションのなをさせることができる
ので、すべての `Stream` のネットワークのためののフォークスレッドプール。

、 `Stream` は、のCPUのなコアのにじて、のくのケースでパフォーマンスをにさせるがあります。

オプションのストリームをのストリームにする

`Optional` の `Stream` をの `Stream` にし、の `Optional` のみをする `Optional` ます。つまり、 `null` なしで、
`Optional.empty()` をしない。

```
Optional<String> op1 = Optional.empty();
Optional<String> op2 = Optional.of("Hello World");

List<String> result = Stream.of(op1, op2)
    .filter(Optional::isPresent)
    .map(Optional::get)
    .collect(Collectors.toList());

System.out.println(result); //[Hello World]
```

ストリームの

すべてのJava `Collection<E>` は、 `Stream<E>` をするための `stream()` および `parallelStream()` メソッドが
あります。

```
Collection<String> stringList = new ArrayList<>();
Stream<String> stringStream = stringList.parallelStream();
```

`Stream<E>` は、の2つのいずれかをしてからできます。

```
String[] values = { "aaa", "bbbb", "ddd", "cccc" };
Stream<String> stringStream = Arrays.stream(values);
Stream<String> stringStreamAlternative = Stream.of(values);
```

`Arrays.stream()` と `Stream.of()` のいは、 `Stream.of()` には `varargs` パラメータがあるため、のようにで
きます。

```
Stream<Integer> integerStream = Stream.of(1, 2, 3);
```

また、できるプリミティブな `Stream` もあります。えば

```
IntStream intStream = IntStream.of(1, 2, 3);
DoubleStream doubleStream = DoubleStream.of(1.0, 2.0, 3.0);
```

これらのプリミティブストリームは、`Arrays.stream()`メソッドをしてすることもできます。

```
IntStream intStream = Arrays.stream(new int[]{ 1, 2, 3 });
```

されたのから`Stream`をすることはです。

```
int[] values= new int[]{1, 2, 3, 4, 5};
IntStream intStream = Arrays.stream(values, 1, 3);
```

のプリミティブストリームは、`boxed`メソッドをして、`boxed`されたタイプのストリームにできます。

```
Stream<Integer> integerStream = intStream.boxed();
```

これは、プリミティブストリームに`Collector`をとってつ`collect`メソッドがないため、データをす
るにもあります。

ストリームチェーンの

これまでのターミナルがびされると、ストリームはじられます。ターミナルだけがしている、の
ストリームをします。すべてのがにされたでしいストリームをするストリームサプライヤをする
ことができます。

```
Supplier<Stream<String>> streamSupplier = () -> Stream.of("apple", "banana", "orange",
"grapes", "melon", "blueberry", "blackberry")
.map(String::toUpperCase).sorted();

streamSupplier.get().filter(s -> s.startsWith("A")).forEach(System.out::println);

// APPLE

streamSupplier.get().filter(s -> s.startsWith("B")).forEach(System.out::println);

// BANANA
// BLACKBERRY
// BLUEBERRY
```

`int[]`はストリームをして`List<Integer>`できます

```
int[] ints = {1,2,3};
List<Integer> list = IntStream.of(ints).boxed().collect(Collectors.toList());
```

ストリームにするの

Javaの8とばれるクラスを[IntSummaryStatistics](#)、[DoubleSummaryStatistics](#)と[LongSummaryStatistics](#)
などのためのオブジェクトをえ`count`、`min`、`max`、`sum`、および`average`。

Java SE 8

```
List<Integer> naturalNumbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
IntSummaryStatistics stats = naturalNumbers.stream()
                                           .mapToInt((x) -> x)
                                           .summaryStatistics();

System.out.println(stats);
```

は

Java SE 8

```
IntSummaryStatistics{count=10, sum=55, min=1, max=10, average=5.500000}
```

ストリームのスライスをする

コレクションの21から50までのものをむ30ののStreamをします。

```
final long n = 20L; // the number of elements to skip
final long maxSize = 30L; // the number of elements the stream should be limited to
final Stream<T> slice = collection.stream().skip(n).limit(maxSize);
```

ノート

- `n`がまたは`maxSize`がの、`IllegalArgumentException`がスローされる
- `skip(long)`と`limit(long)`がです
- ストリームに`n`のがまれている、`skip(n)`はのストリームをします
- `skip(long)`と`limit(long)`は、シーケンシャルストリームパイプラインではなですが、パイプラインではにになります

ストリームをする

の

```
Collection<String> abc = Arrays.asList("a", "b", "c");
Collection<String> digits = Arrays.asList("1", "2", "3");
Collection<String> greekAbc = Arrays.asList("alpha", "beta", "gamma");
```

1 - 2つのStreamする

```
final Stream<String> concat1 = Stream.concat(abc.stream(), digits.stream());

concat1.forEach(System.out::print);
// prints: abc123
```

2 - 3つのStreamする

```
final Stream<String> concat2 = Stream.concat(
    Stream.concat(abc.stream(), digits.stream()),
    greekAbc.stream());
```

```
System.out.println(concat2.collect(Collectors.joining(", ")));
// prints: a, b, c, 1, 2, 3, alpha, beta, gamma
```

あるいはネストされたするために `concat()` `Stream` `s` もとすることができる `flatMap()`。

```
final Stream<String> concat3 = Stream.of(
    abc.stream(), digits.stream(), greekAbc.stream())
    .flatMap(s -> s);
// or `.flatMap(Function.identity());` (java.util.function.Function)

System.out.println(concat3.collect(Collectors.joining(", ")));
// prints: a, b, c, 1, 2, 3, alpha, beta, gamma
```

くされた `Stream` にアクセスすると、いコールチェーンや `StackOverflowException` がするがあるため、りしから `Stream` するときはしてください。

IntStream から String へ

Javaには `Char` ストリームがありません。したがって、`String` て `Stream of Character` をする、`String.codePoints()` メソッドをしてコードポイントの `IntStream` をすることができます。 `IntStream` はのようになります

```
public IntStream stringToIntStream(String in) {
    return in.codePoints();
}
```

つまり、をの、つまり `IntStreamToString` でののはもうしです。それはのようにしてうことができます

```
public String intStreamToString(IntStream intStream) {
    return intStream.collect(StringBuilder::new, StringBuilder::appendCodePoint,
        StringBuilder::append).toString();
}
```

ストリームをしてべえ

```
List<String> data = new ArrayList<>();
data.add("Sydney");
data.add("London");
data.add("New York");
data.add("Amsterdam");
data.add("Mumbai");
data.add("California");

System.out.println(data);

List<String> sortedData = data.stream().sorted().collect(Collectors.toList());

System.out.println(sortedData);
```

```
[Sydney, London, New York, Amsterdam, Mumbai, California]
[Amsterdam, California, London, Mumbai, New York, Sydney]
```

をとするオーバーロード `sorted` バージョンがあるため、なるメカニズムをすることもできます。

また、ソートにラムダをすることもできます。

```
List<String> sortedData2 = data.stream().sorted((s1,s2) ->
s2.compareTo(s1)).collect(Collectors.toList());
```

これは `[Sydney, New York, Mumbai, London, California, Amsterdam]`

`Comparator.reverseOrder()` をすると、けの `reverse` をうコンパレータをできます。

```
List<String> reverseSortedData =
data.stream().sorted(Comparator.reverseOrder()).collect(Collectors.toList());
```

プリミティブのストリーム

Javaは `IntStream int`、`LongStream long`、`DoubleStream double` の3のプリミティブにした `Stream` します。それぞれのプリミティブのされたのに、のためのいくつかのメソッドもします。例えば

```
IntStream is = IntStream.of(10, 20, 30);
double average = is.average().getAsDouble(); // average is 20.0
```

ストリームのをにめる

Analogは、`collect()` によって `Stream` コレクションをするために、`Stream.toArray()` メソッドでを
できます。

```
List<String> fruits = Arrays.asList("apple", "banana", "pear", "kiwi", "orange");

String[] filteredFruits = fruits.stream()
    .filter(s -> s.contains("a"))
    .toArray(String[]::new);

// prints: [apple, banana, pear, orange]
System.out.println(Arrays.toString(filteredFruits));
```

`String[]::new` は、なのメソッドです。コンストラクタです。

とするのの

にする `Stream` のをつけることができます。

このでは、が50000えるの `Integer` をつけるでしょう。

```
IntStream.iterate(1, i -> i + 1) // Generate an infinite stream 1,2,3,4...
```



```
.filter(i -> (i*i) > 50000) // Filter to find elements where the square is >50000
.findFirst(); // Find the first filtered element
```

このはとにOptionalIntをします。

のStream、Javaはをつけるまでのチェックをけます。Stream、Javaがをいたしたにもかかわらずをつけることができないは、のOptionalIntします。

IntStreamをしてインデックスをする

のStreamは、のインデックスにアクセスできません。インデックスにアクセスしながらまたはArrayListをするには、IntStream.range(start, endExclusive)します。

```
String[] names = { "Jon", "Darin", "Bauke", "Hans", "Marc" };

IntStream.range(0, names.length)
    .mapToObj(i -> String.format("#%d %s", i + 1, names[i]))
    .forEach(System.out::println);
```

range(start, endExclusive)メソッドはのIntStreamをし、mapToObj(mapper)はStringストリームをします。

- 1 ジョン
- 2 ダーリン
- 3 Bauke
- 4 ハンズ
- 5 マーク

これは、のforループをカウンタでするのによくていますが、パイプラインとのがあります。

```
for (int i = 0; i < names.length; i++) {
    String newName = String.format("#%d %s", i + 1, names[i]);
    System.out.println(newName);
}
```

flatMapでストリームをする

にストリーミングなアイテムのStreamは、のStreamフラットすることができます。

リストのはのリストにできます。

```
List<String> list1 = Arrays.asList("one", "two");
List<String> list2 = Arrays.asList("three", "four", "five");
List<String> list3 = Arrays.asList("six");
List<String> finalList = Stream.of(list1, list2,
list3).flatMap(Collection::stream).collect(Collectors.toList());
System.out.println(finalList);

// [one, two, three, four, five, six]
```

リストをとしてみマップをリストにすることができます

```
Map<String, List<Integer>> map = new LinkedHashMap<>();
map.put("a", Arrays.asList(1, 2, 3));
map.put("b", Arrays.asList(4, 5, 6));

List<Integer> allValues = map.values() // Collection<List<Integer>>
    .stream()                         // Stream<List<Integer>>
    .flatMap(List::stream)            // Stream<Integer>
    .collect(Collectors.toList());

System.out.println(allValues);
// [1, 2, 3, 4, 5, 6]
```

Map ListをのStreamまとめることができます

```
List<Map<String, String>> list = new ArrayList<>();
Map<String, String> map1 = new HashMap();
map1.put("1", "one");
map1.put("2", "two");

Map<String, String> map2 = new HashMap();
map2.put("3", "three");
map2.put("4", "four");
list.add(map1);
list.add(map2);

Set<String> output = list.stream() // Stream<Map<String, String>>
    .map(Map::values)              // Stream<List<String>>
    .flatMap(Collection::stream)  // Stream<String>
    .collect(Collectors.toSet()); // Set<String>
// [one, two, three, four]
```

ストリームについたをする

キーがしないなケース

```
Stream<String> characters = Stream.of("A", "B", "C");

Map<Integer, String> map = characters
    .collect(Collectors.toMap(element -> element.hashCode(), element -> element));
// map = {65=A, 66=B, 67=C}
```

よりなものにするために、Function インタフェース - [Function.identity\(\)](#) メソッドをすることができます。このlambda `element -> element` を `Function.identity()` きえることができます。

キーがしている

`Collectors.toMap` の [javadoc](#) のはのとおりです。

マッピングされたキーにがまれている `Object.equals(Object)` につて、コレクションがされると `IllegalStateException` がスローされます。マッピングされたキーにがあるは、

わりに`toMap(Function, Function, BinaryOperator)`します。

```
Stream<String> characters = Stream.of("A", "B", "B", "C");

Map<Integer, String> map = characters
    .collect(Collectors.toMap(
        element -> element.hashCode(),
        element -> element,
        (existingVal, newVal) -> (existingVal + newVal)));

// map = {65=A, 66=BB, 67=C}
```

`Collectors.toMap(...)`された`BinaryOperator`は、のにされるをします。できる

- ストリームのものがされるようにいをし、
- ストリームのものがされるようにしいをします。
- いとしいをする

によるグループけ

カスケードされたデータベースの「グループ」とのをするがあるは、`Collectors.groupingBy`をできます。すると、のでは、々のがにマップされたマップがされます。

```
List<Person> people = Arrays.asList(
    new Person("Sam", "Rossi"),
    new Person("Sam", "Verdi"),
    new Person("John", "Bianchi"),
    new Person("John", "Rossi"),
    new Person("John", "Verdi")
);

Map<String, List<String>> map = people.stream()
    .collect(
        // function mapping input elements to keys
        Collectors.groupingBy(Person::getName,
        // function mapping input elements to values,
        // how to store values
        Collectors.mapping(Person::getSurname, Collectors.toList()));
);

// map = {John=[Bianchi, Rossi, Verdi], Sam=[Rossi, Verdi]}
```

イデオンにきる

ストリームをしたランダムなの

WebサービスのセッションIDや、アプリケーションのパスワードなど、ランダムなStringsをすることがあります。これは`Stream`をってにできます。

まず、ジェネレータをするがあります。された`String`のセキュリティをするには、`SecureRandom`をすることをおめします。

`SecureRandom`はにコストがかかるので、だけこれをし、`setSeed()`メソッドの1つをびしてシードす

ることをお`setSeed()` します。

```
private static final SecureRandom rng = new SecureRandom(SecureRandom.generateSeed(20));  
//20 Bytes as a seed is rather arbitrary, it is the number used in the JavaDoc example
```

ランダムな `String` をするとき、例えばとのみのみをします。したがって、で `Stream` をフィルタリングするためにできる `boolean` をすメソッドをできます。

```
//returns true for all chars in 0-9, a-z and A-Z
boolean useThisCharacter(char c){
    //check for range to avoid using all unicode Letter (e.g. some chinese symbols)
    return c >= '0' && c <= 'z' && Character.isLetterOrDigit(c);
}
```

に、RNGをして、`useThisCharacter` チェックをすセットをむのさのランダムなStringをすることができます。

```
public String generateRandomString(long length){
    //Since there is no native CharStream, we use an IntStream instead
    //and convert it to a Stream<Character> using mapToObj.
    //We need to specify the boundaries for the int values to ensure they can safely be cast
    to char
    Stream<Character> randomCharStream = rng.ints(Character.MIN_CODE_POINT,
Character.MAX_CODE_POINT).mapToObj(i -> (char)i).filter(c ->
this::useThisCharacter).limit(length);

    //now we can use this Stream to build a String utilizing the collect method.
    String randomString = randomCharStream.collect(StringBuilder::new, StringBuilder::append,
StringBuilder::append).toString();
    return randomString;
}
```

ストリームをしたの

Stream `s`、に `IntStream` は、 Σ をするエレガントなです。 `Stream` のは、のとしてできます。

例えば、MadhavaのPiのはのでえられます [wikipedia](#)。

$$\pi = \sqrt{12} \sum_{k=0}^{\infty} \frac{(-3)^{-k}}{2k+1} = \sqrt{12} \sum_{k=0}^{\infty} \frac{(-\frac{1}{3})^k}{2k+1} = \sqrt{12} \left(\frac{1}{1 \cdot 3^0} - \frac{1}{3 \cdot 3^1} + \frac{1}{5 \cdot 3^2} - \frac{1}{7 \cdot 3^3} + \cdots \right)$$

これはのでできます。えは、101の

```
double pi = Math.sqrt(12) *
    IntStream.rangeClosed(0, 100)
        .mapToDouble(k -> Math.pow(-3, -1 * k) / (2 * k + 1))
        .sum();
```

double なのは、29のをするだけで、`Math.Pi` とできないをるには `Math.Pi`。

ストーリーおよびメソッドをしたプロセスの

メソッドはれたコードをし、`Stream`メソッドをすると、なプロセスをみやすくすることができます。のコードをえてみましょう

```
public interface Ordered {
    default int getOrder(){
        return 0;
    }
}

public interface Valued<V extends Ordered> {
    boolean hasPropertyTwo();
    V getValue();
}

public interface Thing<V extends Ordered> {
    boolean hasPropertyOne();
    Valued<V> getValuedProperty();
}

public <V extends Ordered> List<V> myMethod(List<Thing<V>> things) {
    List<V> results = new ArrayList<V>();
    for (Thing<V> thing : things) {
        if (thing.hasPropertyOne()) {
            Valued<V> valued = thing.getValuedProperty();
            if (valued != null && valued.hasPropertyTwo()){
                V value = valued.getValue();
                if (value != null){
                    results.add(value);
                }
            }
        }
    }
    results.sort((a, b)->{
        return Integer.compare(a.getOrder(), b.getOrder());
    });
    return results;
}
```

してきえ、このの`Stream`sおよびがはるかにみやすいですし、プロセスのステップをかつにされている-それだけくはない、それはまた、インタフェースとクラスは、ステップでコードをしているでしています。

```
public <V extends Ordered> List<V> myMethod(List<Thing<V>> things) {
    return things.stream()
        .filter(Thing::hasPropertyOne)
        .map(Thing::getValuedProperty)
        .filter(Objects::nonNull)
        .filter(Valued::hasPropertyTwo)
        .map(Valued::getValue)
        .filter(Objects::nonNull)
        .sorted(Comparator.comparing(Ordered::getOrder))
        .collect(Collectors.toList());
}
```

Map.Entryのストリームをしてマッピングのをする

Stream

をするがあるはマップするがありますが、もしたいは、のようなユーティリティメソッドをして `Stream` を `Map.Entry<K, V>` マップできます。

```
public static <K, V> Function<K, Map.Entry<K, V>> entryMapper(Function<K, V> mapper){
    return (k)->new AbstractMap.SimpleEntry<>(k, mapper.apply(k));
}
```

に、コンバータをして、のとマップされたのにアクセスできる `Stream` をできます。

```
Set<K> mySet;
Function<K, V> transformer = SomeClass::transformerMethod;
Stream<Map.Entry<K, V>> entryStream = mySet.stream()
    .map(entryMapper(transformer));
```

その、どおり `Stream` をしけることができます。これにより、コレクションをするオーバーヘッドがされます。

ストリームカテゴリ

ストリームは、との2つのなカテゴリと、ステートレスとステートフルの2つのサブカテゴリにされます。

なは、な `Stream.map` ようにに `Stream.map` 。ストリームがにされるまでびされません。これはにできます

```
Arrays.asList(1, 2, 3).stream().map(i -> {
    throw new RuntimeException("not gonna happen");
    return i;
});
```

は、ストリームののビルディングブロックであり、ソースのにされ、ストリームチェーンをトリガーするターミナルがきます。

ターミナル

ターミナルは、ストリームのをトリガーするものです。よりなのは、 `Stream.forEach` または `Stream.collect` です。らは、ののにされ、ほとんどにしています。

ステートレス

ステートレスとは、アイテムがのアイテムのコンテキストなしでされることをします。ステート

レスでは、ストリームをメモリにできます。ストリームののにするをとしない `Stream.map` や `Stream.filter` ようなは、ステートレスとみなされます。

ステートフルな

ステートフルとは、のがストリームののにすることをします。これには、をするがあります。ステートフル・オペレーションは、い、またはのストリームですることがあります。 `Stream.sorted` ようなでは、がされるにストリームのがされ、いストリームのでされます。これはいストリームでされます あなたのにおいてしてください

```
// works - stateless stream
long BIG_ENOUGH_NUMBER = 999999999;
IntStream.iterate(0, i -> i + 1).limit(BIG_ENOUGH_NUMBER).forEach(System.out::println);
```

これにより、 `Stream.sorted` ステートフルなのためにメモリがします。

```
// Out of memory - stateful stream
IntStream.iterate(0, i -> i +
1).limit(BIG_ENOUGH_NUMBER).sorted().forEach(System.out::println);
```

イテレータをストリームにする

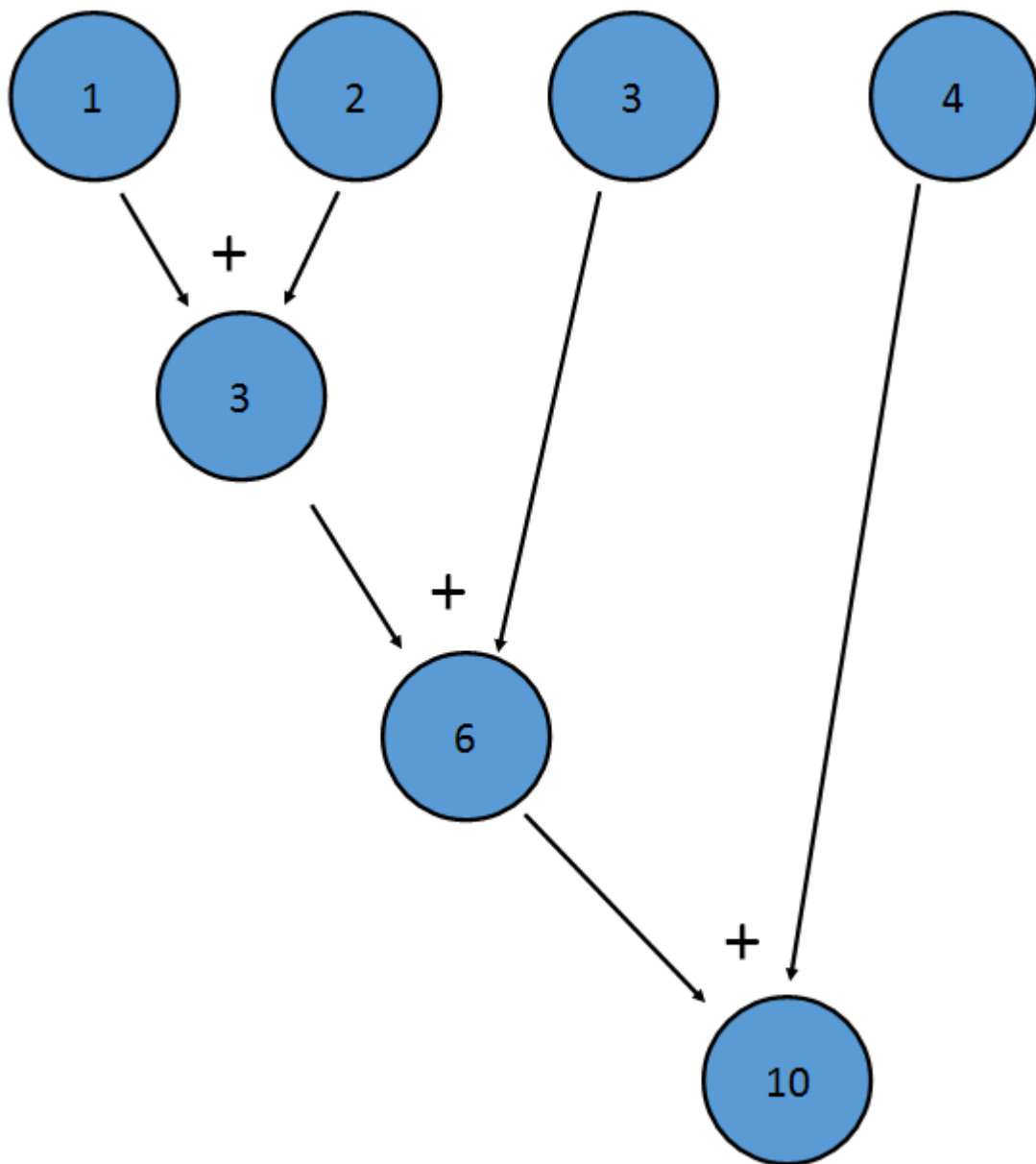
イテレータをストリームにするには、 `Spliterators.splititerator()` または `Spliterators.splititeratorUnknownSize()` をします。

```
Iterator<String> iterator = Arrays.asList("A", "B", "C").iterator();
Spliterator<String> spliterator = Spliterators.splititeratorUnknownSize(iterator, 0);
Stream<String> stream = StreamSupport.stream(spliterator, false);
```

ストリームによる

とは、ストリームのすべてののに2をして1つのにするプロセスです。

`IntStream` の `sum()` メソッドはのです。ストリームのすべてののにをし、なが1になります。



これは、 $((1+2)+3)+4$

ストリームの `reduce` メソッドをすると、カスタムリダクションをできます。 `reduce` メソッドをして `sum()` メソッドをすることはです。

```
IntStream istr;  
  
//Initialize istr  
  
OptionalInt istr.reduce((a,b)->a+b);
```

`Optional` バージョンがされ、のストリームをにできるようになります。

ののは、 `Stream<LinkedList<T>>` をの `LinkedList<T>` することです。


```
Stream<LinkedList<T>> listStream;

//Create a Stream<LinkedList<T>>

Optional<LinkedList<T>> bigList = listStream.reduce((LinkedList<T> list1, LinkedList<T> list2)->{
    LinkedList<T> retList = new LinkedList<T>();
    retList.addAll(list1);
    retList.addAll(list2);
    return retList;
});
```

アイデンティティをすることもできます。たとえば、 $x+0==x$ ように、の、のは0です。の、のは1であり、 $x*1==x$ 。の、のリストをのリストにすると、「する」リストはされないため、identityはのLinkedList<T>です。

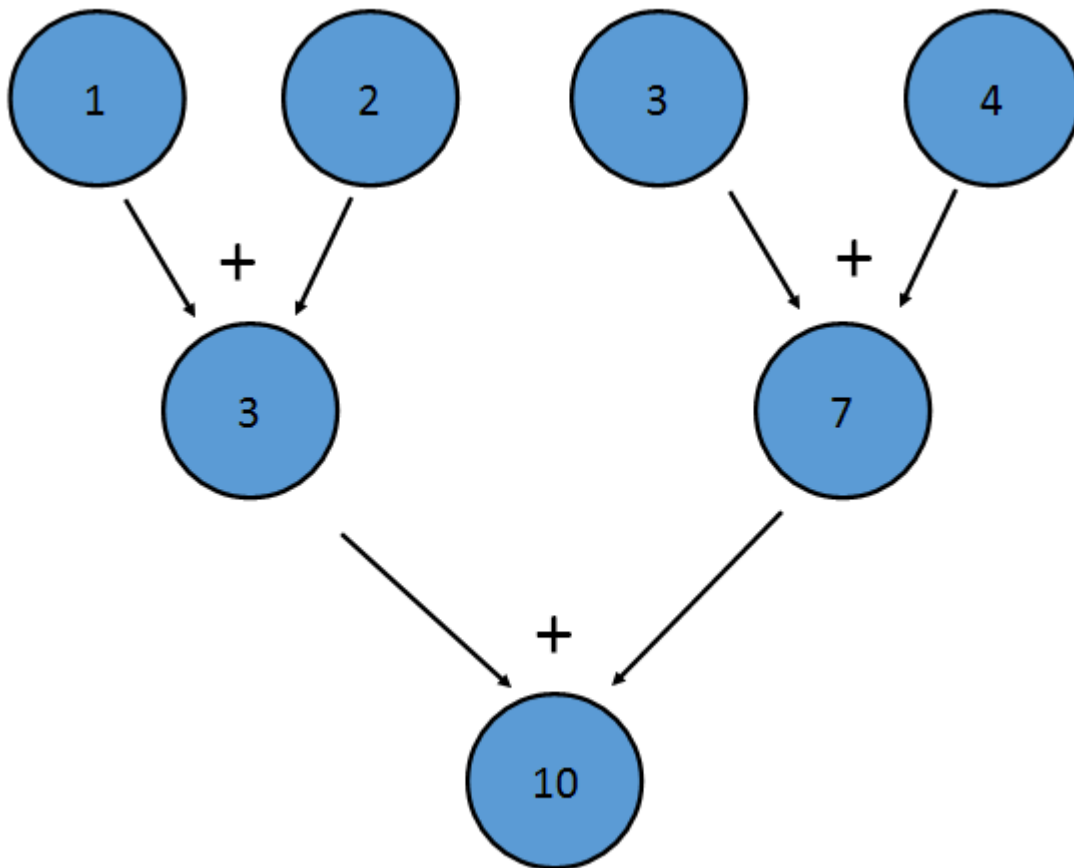
```
Stream<LinkedList<T>> listStream;

//Create a Stream<LinkedList<T>>

LinkedList<T> bigList = listStream.reduce(new LinkedList<T>(), (LinkedList<T> list1,
LinkedList<T> list2)->{
    LinkedList<T> retList = new LinkedList<T>();
    retList.addAll(list1);
    retList.addAll(list2);
    return retList;
});
```

identityがされている、りはのストリームでびされたOptional-ifでラップされないことにしてください。reduce reduce() はidentityをします。

バイナリは、 $(a+b)+c==a+(b+c)$ するでなければなりません。これは、をのでらすことができるためです。えば、のようなのは、のようにできます。



これは、 $((1+2)+(3+4))$ きみとです。また、`Stream`のをプロセッサでらすことができ、にプロセッサのをみわせることができます。

のStringにストリームをする

にされるユースケースは、ストリームをのでったストリームからStringをすることです。ののように、`Collectors.joining()` メソッドをこのためにできます。

```
Stream<String> fruitStream = Stream.of("apple", "banana", "pear", "kiwi", "orange");

String result = fruitStream.filter(s -> s.contains("a"))
    .map(String::toUpperCase)
    .sorted()
    .collect(Collectors.joining(", "));

System.out.println(result);
```

アップル、バナナ、オレンジ、PEAR

`Collectors.joining()` メソッドは、プリフィックスとポストフィックスをすることもできます。

```
String result = fruitStream.filter(s -> s.contains("e"))
    .map(String::toUpperCase)
    .sorted()
```

```
.collect(Collectors.joining(", ", "Fruits: ", "."));  
System.out.println(result);
```

フルーツアップル、オレンジ、PEAR。

[イデオンにきる](#)

[オンラインでストリームをむ](#) <https://riptutorial.com/ja/java/topic/88/ストリーム>

103: セキュリティ

Examples

ハッシュをする

なるアルゴリズムをしてさなデータブロックのハッシュをするには

```
final MessageDigest md5 = MessageDigest.getInstance("MD5");
final MessageDigest sha1 = MessageDigest.getInstance("SHA-1");
final MessageDigest sha256 = MessageDigest.getInstance("SHA-256");

final byte[] data = "FOO BAR".getBytes();

System.out.println("MD5      hash: " + DatatypeConverter.printHexBinary(md5.digest(data)));
System.out.println("SHA1     hash: " + DatatypeConverter.printHexBinary(sha1.digest(data)));
System.out.println("SHA256  hash: " + DatatypeConverter.printHexBinary(sha256.digest(data)));
```

このをします。

```
MD5      hash: E99E768582F6DD5A3BA2D9C849DF736E
SHA1     hash: 0135FAA6323685BA8A8FF8D3F955F0C36949D8FB
SHA256  hash: 8D35C97BCD902B96D1B551741BBE8A7F50BB5A690B4D0225482EAA63DBFB9DED
```

Javaプラットフォームののにじて、のアルゴリズムをできます。

にランダムなデータをする

にランダムなデータのサンプルをするには

```
final byte[] sample = new byte[16];

new SecureRandom().nextBytes(sample);

System.out.println("Sample: " + DatatypeConverter.printHexBinary(sample));
```

のようなをします。

```
Sample: E4F14CEA2384F70B706B53A6DF8C5EFE
```

`nextBytes()` は、されているアルゴリズムによってエントロピーがまるにブロックされることにしてください。

アルゴリズムとプロバイダをするには

```
final byte[] sample = new byte[16];
final SecureRandom randomness = SecureRandom.getInstance("SHA1PRNG", "SUN");
```

```
randomness.nextBytes(sample);

System.out.println("Provider: " + randomness.getProvider());
System.out.println("Algorithm: " + randomness.getAlgorithm());
System.out.println("Sample: " + DatatypeConverter.printHexBinary(sample));
```

のようなをします。

```
Provider: SUN version 1.8
Algorithm: SHA1PRNG
Sample: C80C44BAEB352FD29FBBE20489E4C0B9
```

1のペアをする

なるアルゴリズムとキーサイズをしてキーペアをするには

```
final KeyPairGenerator dhGenerator = KeyPairGenerator.getInstance("DiffieHellman");
final KeyPairGenerator dsaGenerator = KeyPairGenerator.getInstance("DSA");
final KeyPairGenerator rsaGenerator = KeyPairGenerator.getInstance("RSA");

dhGenerator.initialize(1024);
dsaGenerator.initialize(1024);
rsaGenerator.initialize(2048);

final KeyPair dhPair = dhGenerator.generateKeyPair();
final KeyPair dsaPair = dsaGenerator.generateKeyPair();
final KeyPair rsaPair = rsaGenerator.generateKeyPair();
```

のアルゴリズムとキーサイズは、Javaプラットフォームのでできるがあります。

キーをするにするのをするには

```
final KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");

generator.initialize(2048, SecureRandom.getInstance("SHA1PRNG", "SUN"));

final KeyPair pair = generator.generateKeyPair();
```

デジタルのと

をするには

```
final PrivateKey privateKey = keyPair.getPrivate();
final byte[] data = "FOO BAR".getBytes();
final Signature signer = Signature.getInstance("SHA1withRSA");

signer.initSign(privateKey);
signer.update(data);

final byte[] signature = signer.sign();
```

アルゴリズムは、ペアをするためにされるアルゴリズムとがなければならないことにしてください

い。

をするには

```
final PublicKey publicKey = keyPair.getPublic();
final Signature verifier = Signature.getInstance("SHA1withRSA");

verifier.initVerify(publicKey);
verifier.update(data);

System.out.println("Signature: " + verifier.verify(signature));
```

このをします。

```
Signature: true
```

lによるデータの

でデータをするには

```
final Cipher rsa = Cipher.getInstance("RSA");

rsa.init(Cipher.ENCRYPT_MODE, keyPair.getPublic());
rsa.update(message.getBytes());
final byte[] result = rsa.doFinal();

System.out.println("Message: " + message);
System.out.println("Encrypted: " + DatatypeConverter.printHexBinary(result));
```

のようなをします。

```
Message: Hello
Encrypted: 5641FBB9558ECFA9ED...
```

Cipherオブジェクトをするときは、されているキーのタイプとのあるをするがあることにしてください。サポートされているのリストについては、「[JCAアルゴリズム](#)」をしてください。RSAデータの、`message.getBytes()`さはキーのサイズよりさくなければなりません。は、この[SO](#)をしてください。

データをするには

```
final Cipher rsa = Cipher.getInstance("RSA");

rsa.init(Cipher.DECRYPT_MODE, keyPair.getPrivate());
rsa.update(cipherText);
final String result = new String(rsa.doFinal());

System.out.println("Decrypted: " + result);
```

のをします。

Decrypted: Hello

オンラインでセキュリティをむ <https://riptutorial.com/ja/java/topic/7529/セキュリティ->

104: セキュリティ

き

Javaのセキュリティプラクティスは、2つのくにされたカテゴリにけることができます。Javaプラットフォームのセキュリティー、セキュアなJavaプログラミングをします。

Javaプラットフォームのセキュリティプラクティスは、JVMのセキュリティとのにします。JCEプロバイダやセキュリティポリシーなどのトピックがまれています。

セキュアなJavaプログラミングのプラクティスは、セキュアなJavaプログラムをします。これには、やの、などのトピックがまれます。

をにするがありますが、カバーするのあるトピックはのとおりです。

1. JCEプロバイダのコンセプト/
2. リストアイテム

Examples

JCE

Java Cryptography ExtensionJCEは、がプログラムでをかつにできるようにするために、JVMにみまれたフレームワークです。これは、JCEプロバイダのシステムをしてなをにしなから、プログラマにシンプルでポータブルなインターフェイスをすることです。

キーとキー

JCEはとをしますが、にをするのはです。ここでしいををします。

にキーををするためにけられているベストプラクティスの1つは、としてではなく、`byte`としてのみすることです。これは、Javaがであり、メモリで"クリア"または"ゼロ"にすることができないためです。へのはできますが、メモリのセグメントがガベージコレクションされてされるまで、なはメモリにります。は、プログラムのメモリをダンプしてにキーをつけることができるきなウィンドウをっていました。に、`byte`はであり、そのをきすることができます。もはやそれらをとしなくなるとすぐにあなたのキーをゼロにすることはいえです。

なJavaの

コンテンツがです

ネットワーキングの

コンテンツがです

ランダムとあなた

コンテンツがです

ほとんどのアプリケーションでは、`java.util.Random`クラスは「ランダム」なデータのソースです。からランダムなをするか、ランダムなをするか、な "の"をするがあるは、おそらく`Random`します。

しかし、くのシステムはセキュリティのためにランダムにしており、ランダムによってされる`Random`はにではありません。ランダムながなのは、わりに`SecureRandom`をするがあります。

ハッシュと

よりくのがです。

ハッシュは、3つのなをつのクラスのメンバーです。、および。

じデータがえられた、ハッシュはにじをします。すなわち、 $X = Y$ の、 fX はにハッシュ f にして fY にしい。

ハッシュへの2つのは、じをもたらすことはありません。つまり、 X と Y ののにして、 $X = Y$ 、 $fX = fY$ ならば、

ではないにしても、ハッシュを「する」ことはです。すなわち、 fX のみがえられれば、の X が f ブルートフォースをしてすべてのなをとるのをれることはできません。 $f1fX = X$ となるような $f1$ はしてはならない。

くの中には、これらののなくとも1つがありません。たとえば、MD5とSHA1にはがあることがられています。つまり、じをつ2つのです。そのため、がけています。セキュアであるとえられるのには、SHA-256とSHA-512があります。

オンラインでセキュリティをむ <https://riptutorial.com/ja/java/topic/9371/セキュリティ->

105: セット

Examples

をつHashSetをする

HashSetからするしいクラスをすることができます

```
Set<String> h = new HashSet<String>() {{
    add("a");
    add("b");
}};
```

ワンラインソリューション

```
Set<String> h = new HashSet<String>(Arrays.asList("a", "b"));
```

グアバをする

```
Sets.newHashSet("a", "b", "c")
```

ストリームの

```
Set<String> set3 = Stream.of("a", "b", "c").collect(toSet());
```

セットのタイプと

に、はのをするコレクションののです。は、 equals() および hashCode() メソッドによってされます。

ソートはセットのタイプによってまります。

HashSet - ランダムソート

Java SE 7

```
Set<String> set = new HashSet<> ();
set.add("Banana");
set.add("Banana");
set.add("Apple");
set.add("Strawberry");

// Set Elements: ["Strawberry", "Banana", "Apple"]
```

LinkedHashSet

- オーダー

Java SE 7

```
Set<String> set = new LinkedHashSet<> ();
set.add("Banana");
set.add("Banana");
set.add("Apple");
set.add("Strawberry");

// Set Elements: ["Banana", "Apple", "Strawberry"]
```

TreeSet **compareTo()** **または** **Comparator**

Java SE 7

```
Set<String> set = new TreeSet<> ();
set.add("Banana");
set.add("Banana");
set.add("Apple");
set.add("Strawberry");

// Set Elements: ["Apple", "Banana", "Strawberry"]
```

Java SE 7

```
Set<String> set = new TreeSet<> ((string1, string2) -> string2.compareTo(string1));
set.add("Banana");
set.add("Banana");
set.add("Apple");
set.add("Strawberry");

// Set Elements: ["Strawberry", "Banana", "Apple"]
```

Setは、するをむことができないコレクションです。それは、をモデルする。

Setは、 HashSet 、 TreeSet 、 LinkedHashSetなどのさまざまなクラスでされています。

例えば

HashSet

```
Set<T> set = new HashSet<T>();
```

ここで、TはString、Integerまたはのオブジェクトです。 **HashSet**はO1のクイックルックアップをにしますが、されたデータをソートせず、アイテムのをいます。

TreeSet

これは、`Olgn`をるなのをにして、ソートされたでデータをします。アイテムのはされません。

```
TreeSet<T> sortedSet = new TreeSet<T>();
```

LinkedHashSet

これは、`HashSet`リンクされたリストののです。されたでアイテムをすることができます。そのはソートされていません。O1のがされていますが、バッキングリンクリストのには`HashSet`よりもコストがかかります。

```
LinkedHashSet<T> linkedhashset = new LinkedHashSet<T>();
```

セットの

セットとはですか

は、の2つのがしくないなをつのをむデータである。

セットのタイプ

1. **HashSet** ハッシュテーブルには`HashMap`インスタンスをにしたセットです。
2. リンクされた**HashSet**なで、ハッシュテーブルとリンクされたリストにちされたセット
3. **TreeSet** `TreeMap`につく `NavigableSet`ののです。

セットの

```
Set<Integer> set = new HashSet<Integer>(); // Creates an empty Set of Integers

Set<Integer> linkedHashSet = new LinkedHashSet<Integer>(); //Creates a empty Set of Integers,
with predictable iteration order
```

セットにをする

`add()` メソッドをしてをセットにすることができます

```
set.add(12); // - Adds element 12 to the set
set.add(13); // - Adds element 13 to the set
```

このメソッドをしたのセット

```
set = [12,13]
```

セットのすべてのをする

```
set.clear(); //Removes all objects from the collection.
```

このセットの

```
set = []
```

がセットのかどうかをチェックする

セットののは、 `contains()` メソッドを使ってチェックすることができます

```
set.contains(0); //Returns true if a specified object is an element within the set.
```

False

セットがであるかどうかをする

`isEmpty()` メソッドをして、Setがであるかどうかをできます。

```
set.isEmpty(); //Returns true if the set has no elements
```

True

セットからをする

```
set.remove(0); // Removes first occurrence of a specified object from the collection
```

セットのサイズをする

```
set.size(); //Returns the number of elements in the collection
```

0

のセットからリストをする

しいリストの

```
List<String> list = new ArrayList<String>(listOfElements);
```

List.addAll メソッドの

```
Set<String> set = new HashSet<String>();  
set.add("foo");  
set.add("boo");  
  
List<String> list = new ArrayList<String>();  
list.addAll(set);
```

Java 8 Stream APIの

```
List<String> list = set.stream().collect(Collectors.toList());
```

Setをしてをする

コレクション`elements`があり、じをみ、すべてののがされたのコレクションをします。

```
Collection<Type> noDuplicates = new HashSet<Type>(elements);
```

```
List<String> names = new ArrayList<>(  
    Arrays.asList("John", "Marco", "Jenny", "Emily", "Jenny", "Emily", "John"));  
Set<String> noDuplicates = new HashSet<>(names);  
System.out.println("noDuplicates = " + noDuplicates);
```

```
noDuplicates = [Marco, Emily, John, Jenny]
```

オンラインでセットをむ <https://riptutorial.com/ja/java/topic/3102/セット>

106: ソケット

き

ソケットは、ネットワークでされている2つのプログラムのリンクの1つのエンドポイントです。

Examples

ソケットからみる

```
String hostName = args[0];
int portNumber = Integer.parseInt(args[1]);

try (
    Socket echoSocket = new Socket(hostName, portNumber);
    PrintWriter out =
        new PrintWriter(echoSocket.getOutputStream(), true);
    BufferedReader in =
        new BufferedReader(
            new InputStreamReader(echoSocket.getInputStream()));
    BufferedReader stdIn =
        new BufferedReader(
            new InputStreamReader(System.in))
) {
    //Use the socket
}
```

オンラインでソケットをむ <https://riptutorial.com/ja/java/topic/9918/ソケット>

107: タイプ

- `TargetType target =SourceTypeSource;`

Examples

プリミティブキャストイング

`boolean`はのプリミティブとのでキャストできません。

Unicodeでされたコードポイントマッピングをすると、のとので`char`をキャストできます。 `char`はメモリでなし16ビット2バイトとしてされるため、 `byte` 1バイトへのキャストはこれらのビットのうち8ビットをしますこれはASCIIではです。 `Character`クラスのユーティリティメソッドは、 `int` 4バイトをしてコードポイントとのでやりとりしますが、Unicodeコードポイントをするには`short` 2バイトでもです。

```
int badInt    = (int)  true; // Compiler error: incompatible types

char char1    = (char)   65; // A
byte byte1    = (byte)   'A'; // 65
short short1  = (short)  'A'; // 65
int int1      = (int)    'A'; // 65

char char2    = (char)  8253; // ?
byte byte2    = (byte)   '?'; // 61 (truncated code-point into the ASCII range)
short short2  = (short)  '?'; // 8253
int int2      = (int)    '?'; // 8253
```

プリミティブキャストイング

プリミティブは2つのでキャストできます。 のキャストは、ソースタイプのターゲットタイプのがいにします。

```
//Implicit casting
byte byteVar = 42;
short shortVar = byteVar;
int intVar = shortVar;
long longVar = intVar;
float floatVar = longVar;
double doubleVar = floatVar;
```

キャストイングは、ソース・タイプのがターゲット・タイプよりもきいにするがあります。

```
//Explicit casting
double doubleVar = 42.0d;
float floatVar = (float) doubleVar;
long longVar = (long) floatVar;
int intVar = (int) longVar;
short shortVar = (short) intVar;
```



```
byte byteVar = (byte) shortVar;
```

プリミティブ `float`、`double` をプリミティブにキャストするとき、そのはりてられます。

オブジェクトキャスト

プリミティブとに、オブジェクトはにもにもキャストできます。

のキャストは、ソース・タイプがターゲット・タイプスーパークラスまたはインターフェースへのキャストをまたはするときします。

なキャストは、ソース・タイプがターゲット・タイプサブタイプにキャストによってまたはされているにするがあります。これにより、キャストされるオブジェクトがターゲットタイプまたはターゲットのサブタイプでないに、ランタイム `ClassCastException` がされるがあります。

```
Float floatVar = new Float(42.0f);
Number n = floatVar;                //Implicit (Float implements Number)
Float floatVar2 = (Float) n;        //Explicit
Double doubleVar = (Double) n;     //Throws exception (the object is not Double)
```

な

```
static void testNumericPromotion() {

    char char1 = 1, char2 = 2;
    short short1 = 1, short2 = 2;
    int int1 = 1, int2 = 2;
    float float1 = 1.0f, float2 = 2.0f;

    // char1 = char1 + char2;        // Error: Cannot convert from int to char;
    // short1 = short1 + short2;     // Error: Cannot convert from int to short;
    int1 = char1 + char2;           // char is promoted to int.
    int1 = short1 + short2;         // short is promoted to int.
    int1 = char1 + short2;          // both char and short promoted to int.
    float1 = short1 + float2;       // short is promoted to float.
    int1 = int1 + int2;             // int is unchanged.

}
```

`instanceof`をしてオブジェクトをキャストできるかどうかのテスト

Javaは、オブジェクトがのかそののサブクラスかをテストするために `instanceof` をし `instanceof` 。プログラムは、そのオブジェクトをキャストするか、キャストしないかをできます。

```
Object obj = Calendar.getInstance();
long time = 0;

if(obj instanceof Calendar)
{
    time = ((Calendar)obj).getTime();
}
if(obj instanceof Date)
```

```
{  
    time = ((Date)obj).getTime(); // This line will never be reached, obj is not a Date type.  
}
```

オンラインでタイプをむ <https://riptutorial.com/ja/java/topic/1392/タイプ>

108: デキューインタフェース

き

Dequeは、でのとをサポートするコレクションです。

デュークは「ダブルエンドキュー」ので、「デッキ」とされます。

のDequeでは、のにはありませんが、このインターフェイスはのあるデクをサポートしていますが、サイズがされていないものもサポートしています。

Dequeインタフェースは、StackとQueueのよりもなデータです。これは、スタックとキューのを
にするためです。

ジェネリックはDequeでできます。

```
Deque<Object> deque = new LinkedList<Object>();
```

デキューをキューとしてすると、FIFOれしのがします。

Dequeは、LIFOLast-In-First-Outスタックとしてもできます。

メソッドのについては、 [このドキュメント](#)をしてください。

Examples

Dequeにをする

```
Deque deque = new LinkedList();

//Adding element at tail
deque.add("Item1");

//Adding element at head
deque.addFirst("Item2");

//Adding element at tail
deque.addLast("Item3");
```

Dequeからの

```
//Retrieves and removes the head of the queue represented by this deque
Object headItem = deque.remove();

//Retrieves and removes the first element of this deque.
Object firstItem = deque.removeFirst();
```

```
//Retrieves and removes the last element of this deque.  
Object lastItem = deque.removeLast();
```

せずにをする

```
//Retrieves, but does not remove, the head of the queue represented by this deque  
Object headItem = deque.element();  
  
//Retrieves, but does not remove, the first element of this deque.  
Object firstItem = deque.getFirst();  
  
//Retrieves, but does not remove, the last element of this deque.  
Object lastItem = deque.getLast();
```

Dequeをする

```
//Using Iterator  
Iterator iterator = deque.iterator();  
while(iterator.hasNext()){  
    String Item = (String) iterator.next();  
}  
  
//Using For Loop  
for(Object object : deque) {  
    String Item = (String) object;  
}
```

オンラインでデキューインタフェースをむ <https://riptutorial.com/ja/java/topic/10156/デキューインタフェース>

109: デフォルトメソッド

き

Java 8でされたデフォルトメソッドは、がこのインタフェースののをすることなくしいメソッドをインタフェースにすることをにします。インタフェースをするクラスがそのメソッドのをできないときに、デフォルトとしてされるをインタフェースができるようにするをします。

- パブリックのデフォルト `void methodName{/ * method body * /}`

デフォルトのメソッド

- のサブクラスでにすることなく、をするために、インタフェースでできます。
- サブクラスまたはサブインターフェースによってオーバーライドできます。
- `java.lang.Object`クラスのメソッドをオーバーライドすることはできません。
- のインタフェースをしているクラスが、それぞれのインタフェースからじメソッドシグネチャをつデフォルトメソッドをするは、ののとしてデフォルトメソッドでないかのようにのインタフェースをオーバーライドしてするがあります。
- のをすることなくをすることをしていますが、しくされたデフォルトメソッドとじメソッドシグニチャーをつメソッドをつのサブクラスはとしてします。しかし、スーパークラスにインスタンスメソッドをしたでも、これはてはまります。

メソッド

- インターフェイスでできます。にデフォルトメソッドのユーティリティメソッドとしてされます。
- サブクラスまたはサブインターフェースそれらにされているによってオーバーライドすることはできません。しかし、でもメソッドののに、クラスまたはインタフェースはのものをすることがあります。
- `java.lang.Object`クラスのインスタンスメソッドをオーバーライドすることはできませんでサブクラスのもです。

は、サブクラスとスーパークラスのをしたです。

-	SUPER_CLASS-INSTANCE-METHOD	スーパークラスメソッド
SUB_CLASS-INSTANCE-METHOD	オーバーライド	- コンパイルエラー
SUB_CLASS-STATIC-METHOD	- コンパイルエラー	す

は、インタフェースとクラスのをしたです。

-	インタフェースメソッド	インターフェースメソッド
IMPL_CLASS-INSTANCE-METHOD	オーバーライド	す
IMPL_CLASS-STATIC-METHOD	- コンパイルエラー	す

- <http://www.journaldev.com/2752/java-8-interface-changes-static-method-default-method>
- <https://docs.oracle.com/javase/tutorial/java/landl/override.html>

Examples

デフォルトメソッドのない

```

/**
 * Interface with default method
 */
public interface Printable {
    default void printString() {
        System.out.println( "default implementation" );
    }
}

/**
 * Class which falls back to default implementation of {@link #printString()}
 */
public class WithDefault
    implements Printable
{
}

/**
 * Custom implementation of {@link #printString()}
 */
public class OverrideDefault
    implements Printable {

```

```

@Override
public void printString() {
    System.out.println( "overridden implementation" );
}
}

```

のステートメント

```

new WithDefault().printString();
new OverrideDefault().printString();

```

このをします

default implementation
overridden implementation

デフォルトメソッドのインタフェースメソッドへのアクセス

デフォルトメソッドからのインタフェースメソッドにアクセスすることもできます。

```

public interface Summable {
    int getA();

    int getB();

    default int calculateSum() {
        return getA() + getB();
    }
}

public class Sum implements Summable {
    @Override
    public int getA() {
        return 1;
    }

    @Override
    public int getB() {
        return 2;
    }
}

```

のは3をします

```

System.out.println(new Sum().calculateSum());

```

デフォルトのメソッドは、インターフェイスのメソッドとにすることもできます。

```

public interface Summable {
    static int getA() {
        return 1;
    }

    static int getB() {

```

```

        return 2;
    }

    default int calculateSum() {
        return getA() + getB();
    }
}

public class Sum implements Summable {}

```

のステートメントは、3をします。

```
System.out.println(new Sum().calculateSum());
```

オーバーライドされたデフォルトメソッドにクラスをすることからアクセスする

クラスでは、`super.foo()`はスーパークラスのみをします。`super`インタフェースからデフォルトのものをひすは、インタフェース`Fooable.super.foo()` `super`をするがあります。

```

public interface Fooable {
    default int foo() {return 3;}
}

public class A extends Object implements Fooable {
    @Override
    public int foo() {
        //return super.foo() + 1; //error: no method foo() in java.lang.Object
        return Fooable.super.foo() + 1; //okay, returns 4
    }
}

```

なぜデフォルトメソッドをするのですか

なえは、のをることなくのインターフェースをさせることができるということです。

たとえば、20にした`Swim`インターフェイスがあります。

```

public interface Swim {
    void backStroke();
}

```

たちはすばらしいをしてくれました。たちのインターフェースはにしています。のくのがあり、ソースコードをすることはできません。

```

public class FooSwimmer implements Swim {
    public void backStroke() {
        System.out.println("Do backstroke");
    }
}

```

20には、しいをインターフェースにすることにめましたが、インターフェースがされているよう

にえます。のをるためです。

いJava 8では、 **Defaultメソッド**というしいがされています。

Swimインターフェイスにしいメソッドをできるようになりました。

```
public interface Swim {
    void backStroke();
    default void sideStroke() {
        System.out.println("Default sidestroke implementation. Can be overridden");
    }
}
```

、インタフェースののはすべてします。しかし、もなのは、しくされたメソッドをのにできることです。

こののの1つ、そしてそのの1つは、Java Collectionsフレームワークです。Oracleは、Iterableをしたのコードをすべてすることなく、のIterableインタフェースにforeachメソッドをできませんでした。デフォルトのメソッドをすることで、のIterableのはデフォルトのをします。

クラス、クラス、およびインターフェイスメソッドの

をむクラスのは、すべてのインタフェースのデフォルトよりもされます。

- クラスメソッドは**インターフェイスデフォルトメソッド**よりもされます。

```
public interface Swim {
    default void backStroke() {
        System.out.println("Swim.backStroke");
    }
}

public abstract class AbstractSwimmer implements Swim {
    public void backStroke() {
        System.out.println("AbstractSwimmer.backStroke");
    }
}

public class FooSwimmer extends AbstractSwimmer {
}
```

のステートメント

```
new FooSwimmer().backStroke();
```

する

```
AbstractSwimmer.backStroke
```

- クラスメソッドは**インターフェイスのデフォルトメソッド**よりもされます

```
public interface Swim {
    default void backStroke() {
        System.out.println("Swim.backStroke");
    }
}

public abstract class AbstractSwimmer implements Swim {
}

public class FooSwimmer extends AbstractSwimmer {
    public void backStroke() {
        System.out.println("FooSwimmer.backStroke");
    }
}
```

のステートメント

```
new FooSwimmer().backStroke();
```

する

```
FooSwimmer.backStroke
```

デフォルト メソッドの

のをえてみましょう。

```
public interface A {
    default void foo() { System.out.println("A.foo"); }
}

public interface B {
    default void foo() { System.out.println("B.foo"); }
}
```

じシグニチャをつ default メソッド foo をする2つのインタフェースがあり default 。

これらののインタフェースをしいインタフェースで extend しようと extend と、2つのインタフェースをするがあります。なぜならJavaはこのをにするがあるからです。

まず、メソッド foo を abstract とじシグネチャですと、A と B がオーバーライドされます。

```
public interface ABExtendsAbstract extends A, B {
    @Override
    void foo();
}
```

class ABExtendsAbstract を implement ABExtendsAbstract は、foo をするがあります

```
public class ABExtendsAbstractImpl implements ABExtendsAbstract {
    @Override
    public void foo() { System.out.println("ABImpl.foo"); }
}
```

```
}
```

それとも、あなたはにしいすることができ`default`を。また、[クラスのからオーバーライドされたデフォルトメソッドにアクセス](#)することによって、`A`および`B` `foo`メソッドのコードをすることもできます。

```
public interface ABExtends extends A, B {  
    @Override  
    default void foo() { System.out.println("ABExtends.foo"); }  
}
```

`class ABExtends` `implements ABExtends`は、`foo`をするはあり`not`

```
public class ABExtendsImpl implements ABExtends {}
```

[オンラインでデフォルトメソッドをむ](https://riptutorial.com/ja/java/topic/113/デフォルトメソッド) <https://riptutorial.com/ja/java/topic/113/デフォルトメソッド>

110: ネットワーキング

- しいソケット "localhost"、1234; //アドレス "localhost"とポート1234のサーバにします。
- しいSocketServer "localhost"、1234; //アドレスlocalhostとポート1234にしいソケットをできるソケットサーバーをします。
- socketServer.accept; //クライアントとのにできるしいSocketオブジェクトをける

Examples

ソケットをしたなクライアントとサーバーの

サーバーし、をつ

```
//Open a listening "ServerSocket" on port 1234.
ServerSocket serverSocket = new ServerSocket(1234);

while (true) {
    // Wait for a client connection.
    // Once a client connected, we get a "Socket" object
    // that can be used to send and receive messages to/from the newly
    // connected client
    Socket clientSocket = serverSocket.accept();

    // Here we'll add the code to handle one specific client.
}
```

サーバークライアントの

のスレッドでクライアントをし、のクライアントがにサーバーとやりとりできるようにします。このテクニックは、クライアントのがないOSアーキテクチャーとスレッドのされるにじて、クライアントが1000のりにします。

```
new Thread(() -> {
    // Get the socket's InputStream, to read bytes from the socket
    InputStream in = clientSocket.getInputStream();
    // wrap the InputStream in a reader so you can read a String instead of bytes
    BufferedReader reader = new BufferedReader(
        new InputStreamReader(in, StandardCharsets.UTF_8));
    // Read text from the socket and print line by line
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
}).start();
```

クライアントサーバーにしてメッセージをする

```
// 127.0.0.1 is the address of the server (this is the localhost address; i.e.
// the address of our own machine)
// 1234 is the port that the server will be listening on
Socket socket = new Socket("127.0.0.1", 1234);

// Write a string into the socket, and flush the buffer
OutputStream outputStream = socket.getOutputStream();
PrintWriter writer = new PrintWriter(
    new OutputStreamWriter(outputStream, StandardCharsets.UTF_8));
writer.println("Hello world!");
writer.flush();
```

ソケットをじてをする

のは、みやすくするためにいくつかのをしています。

1. ファイルやそののリソースと、OSをしたらOSにえることができます。ソケットがわったら、`socket.close()` をびしてしくじます。
2. ソケットは、さまざまにするI/O/をします。たとえば、がされるとしたらどうでしょう
ネットワークエラーがしたはどうなりますかこれらのことはたちのコントロールのをえています。このため、くのソケットでがスローされることがありますに `IOException`。

したがって、クライアントのためのよりなコードは、のようなものになります。

```
// "try-with-resources" will close the socket once we leave its scope
try (Socket socket = new Socket("127.0.0.1", 1234)) {
    OutputStream outputStream = socket.getOutputStream();
    PrintWriter writer = new PrintWriter(
        new OutputStreamWriter(outputStream, StandardCharsets.UTF_8));
    writer.println("Hello world!");
    writer.flush();
} catch (IOException e) {
    //Handle the error
}
```

なサーバーとクライアント - な

サーバ

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.net.Socket;
import java.nio.charset.StandardCharsets;

public class Server {
    public static void main(String args[]) {
        try (ServerSocket serverSocket = new ServerSocket(1234)) {
            while (true) {
```

```

        // Wait for a client connection.
        Socket clientSocket = serverSocket.accept();

        // Create and start a thread to handle the new client
        new Thread(() -> {
            try {
                // Get the socket's InputStream, to read bytes
                // from the socket
                InputStream in = clientSocket.getInputStream();
                // wrap the InputStream in a reader so you can
                // read a String instead of bytes
                BufferedReader reader = new BufferedReader(
                    new InputStreamReader(in, StandardCharsets.UTF_8));
                // Read from the socket and print line by line
                String line;
                while ((line = reader.readLine()) != null) {
                    System.out.println(line);
                }
            }
            catch (IOException e) {
                e.printStackTrace();
            }
            finally {
                // This finally block ensures the socket is closed.
                // A try-with-resources block cannot be used because
                // the socket is passed into a thread, so it isn't
                // created and closed in the same block
                try {
                    clientSocket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }).start();
    }
}
catch (IOException e) {
    e.printStackTrace();
}
}
}

```

クライアント

```

import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.Socket;
import java.nio.charset.StandardCharsets;

public class Client {
    public static void main(String args[]) {
        try (Socket socket = new Socket("127.0.0.1", 1234)) {
            // We'll reach this code once we've connected to the server

            // Write a string into the socket, and flush the buffer
            OutputStream outputStream = socket.getOutputStream();
            PrintWriter writer = new PrintWriter(
                new OutputStreamWriter(outputStream, StandardCharsets.UTF_8));

```

```

        writer.println("Hello world!");
        writer.flush();
    } catch (IOException e) {
        // Exception should be handled.
        e.printStackTrace();
    }
}
}

```

InputStreamからのTrustStoreおよびKeyStoreのロード

```

public class TrustLoader {

    public static void main(String args[]) {
        try {
            //Gets the inputstream of a a trust store file under ssl/rpgrenadesClient.jks
            //This path refers to the ssl folder in the jar file, in a jar file in the
            same directory
            //as this jar file, or a different directory in the same directory as the jar
            file
            InputStream stream =
TrustLoader.class.getResourceAsStream("/ssl/rpgrenadesClient.jks");
            //Both trustStores and keyStores are represented by the KeyStore object
            KeyStore trustStore = KeyStore.getInstance(KeyStore.getDefaultType());
            //The password for the trustStore
            char[] trustStorePassword = "password".toCharArray();
            //This loads the trust store into the object
            trustStore.load(stream, trustStorePassword);

            //This is defining the SSLContext so the trust store will be used
            //Getting default SSLContext to edit.
            SSLContext context = SSLContext.getInstance("SSL");
            //TrustMangers hold trust stores, more than one can be added
            TrustManagerFactory factory =
TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
            //Adds the truststore to the factory
            factory.init(trustStore);
            //This is passed to the SSLContext init method
            TrustManager[] managers = factory.getTrustManagers();
            context.init(null, managers, null);
            //Sets our new SSLContext to be used.
            SSLContext.setDefault(context);
        } catch (KeyStoreException | IOException | NoSuchAlgorithmException
            | CertificateException | KeyManagementException ex) {
            //Handle error
            ex.printStackTrace();
        }
    }
}

```

KeyStoreのは、オブジェクトのTrustをKeyきえることをいってじです。さらに、KeyManager[]をSSLContext.initのにすがあります。これはSSLContext.init(keyMangers, trustMangers, null)

ソケットの。なソケットをってWebページをむ

```

import java.io.*;
import java.net.Socket;

public class Main {

    public static void main(String[] args) throws IOException { //We don't handle Exceptions in
this example
        //Open a socket to stackoverflow.com, port 80
        Socket socket = new Socket("stackoverflow.com",80);

        //Prepare input, output stream before sending request
        OutputStream outputStream = socket.getOutputStream();
        InputStream inputStream = socket.getInputStream();
        BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream));
        PrintWriter writer = new PrintWriter(new BufferedOutputStream(outputStream));

        //Send a basic HTTP header
        writer.print("GET / HTTP/1.1\nHost:stackoverflow.com\n\n");
        writer.flush();

        //Read the response
        System.out.println(readFully(reader));

        //Close the socket
        socket.close();
    }

    private static String readFully(Reader in) {
        StringBuilder sb = new StringBuilder();
        int BUFFER_SIZE=1024;
        char[] buffer = new char[BUFFER_SIZE]; // or some other size,
        int charsRead = 0;
        while ( (charsRead = rd.read(buffer, 0, BUFFER_SIZE)) != -1) {
            sb.append(buffer, 0, charsRead);
        }
    }
}

```

のHTTPをすHTTP/1.1 200 OKでまり、HTMLヘッダーのりがき、それにHTMLののWebページがきます。

すぎるEOFをぐには、`readFully()`メソッドがであることにしてください。Webページののにがないと、のわりをらせることができます。そうすれば、`readLine()`はでむか、[Apacheのユーティリティメソッド](#)をするがあります。[io IOtils](#)

このは、ソケットをしてのリソースにするなデモをしています、のWebページにアクセスするではありません。JavaをしてWebページにアクセスするがあるは、[ApacheのHTTPクライアント](#)や[GoogleのHTTPクライアント](#)などのHTTPクライアントライブラリをすることをおめします

UDPデータグラムをしたなクライアント/サーバ

Client.java

```

import java.io.*;
import java.net.*;

```



```

public class Client{
    public static void main(String [] args) throws IOException{
        DatagramSocket clientSocket = new DatagramSocket();
        InetAddress address = InetAddress.getByName(args[0]);

        String ex = "Hello, World!";
        byte[] buf = ex.getBytes();

        DatagramPacket packet = new DatagramPacket(buf,buf.length, address, 4160);
        clientSocket.send(packet);
    }
}

```

この、 `args[0]` をしてサーバのアドレスをします。しているポートは4160です。

Server.java

```

import java.io.*;
import java.net.*;

public class Server{
    public static void main(String [] args) throws IOException{
        DatagramSocket serverSocket = new DatagramSocket(4160);

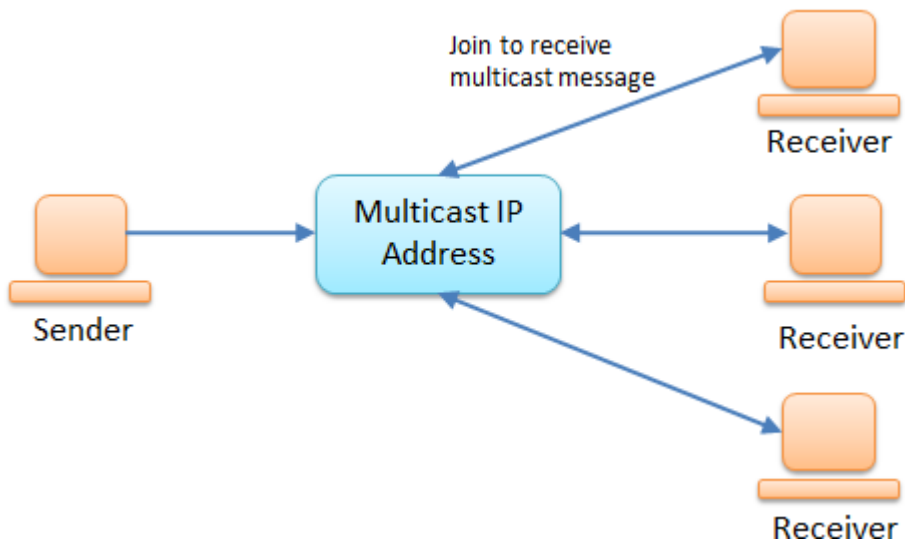
        byte[] rbuf = new byte[256];
        DatagramPacket packet = new DatagramPacket(rbuf, rbuf.length);
        serverSocket.receive(packet);
        String response = new String(packet.getData());
        System.out.println("Response: " + response);
    }
}

```

サーバーで、メッセージをしたじポート4160にDatagramSocketをし、をちます。

マルチキャストینگ

マルチキャストは、Datagram Socketのです。のデータグラムとはなり、マルチキャストینگは々のクライアントをにするのではなく、1つのIPアドレスにし、しているすべてのクライアントがメッセージをします。



サーバーのコード

```
public class Server {  
  
    private DatagramSocket serverSocket;  
  
    private String ip;  
  
    private int port;  
  
    public Server(String ip, int port) throws SocketException, IOException{  
        this.ip = ip;  
        this.port = port;  
        // socket used to send  
        serverSocket = new DatagramSocket();  
    }  
  
    public void send() throws IOException{  
        // make datagram packet  
        byte[] message = ("Multicasting...").getBytes();  
        DatagramPacket packet = new DatagramPacket(message, message.length,  
            InetAddress.getByName(ip), port);  
        // send packet  
        serverSocket.send(packet);  
    }  
  
    public void close(){  
        serverSocket.close();  
    }  
}
```

クライアントのコード

```
public class Client {  
  
    private MulticastSocket socket;  
  
    public Client(String ip, int port) throws IOException {  
  
        // important that this is a multicast socket  
        socket = new MulticastSocket(port);  

```

```

        // join by ip
        socket.joinGroup(InetAddress.getByName(ip));
    }

    public void printMessage() throws IOException{
        // make datagram packet to recieve
        byte[] message = new byte[256];
        DatagramPacket packet = new DatagramPacket(message, message.length);

        // recieve the packet
        socket.receive(packet);
        System.out.println(new String(packet.getData()));
    }

    public void close(){
        socket.close();
    }
}

```

サーバーをするためのコード

```

public static void main(String[] args) {
    try {
        final String ip = args[0];
        final int port = Integer.parseInt(args[1]);
        Server server = new Server(ip, port);
        server.send();
        server.close();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}

```

クライアントをするためのコード

```

public static void main(String[] args) {
    try {
        final String ip = args[0];
        final int port = Integer.parseInt(args[1]);
        Client client = new Client(ip, port);
        client.printMessage();
        client.close();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}

```

にクライアントをするパケットのをするに、クライアントはIPにしているがあります。サーバーをして `send()` メソッドをびし、クライアント `printMessage()` びしますをする。メッセージのにクライアントがされているため、もこりません。

SSLをににするテスト

またはテストで、SSLチェーンがにされていないまだがあります。

とテストをするには、「すべてする」トラストマネージャをインストールすることで、プログラムによるSSLをにすることができます。

```
try {
    // Create a trust manager that does not validate certificate chains
    TrustManager[] trustAllCerts = new TrustManager[] {
        new X509TrustManager() {
            public X509Certificate[] getAcceptedIssuers() {
                return null;
            }
            public void checkClientTrusted(X509Certificate[] certs, String authType) {
            }
            public void checkServerTrusted(X509Certificate[] certs, String authType) {
            }
        }
    };

    // Install the all-trusting trust manager
    SSLContext sc = SSLContext.getInstance("SSL");
    sc.init(null, trustAllCerts, new java.security.SecureRandom());
    HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());

    // Create all-trusting host name verifier
    HostnameVerifier allHostsValid = new HostnameVerifier() {
        public boolean verify(String hostname, SSLSession session) {
            return true;
        }
    };

    // Install the all-trusting host verifier
    HttpsURLConnection.setDefaultHostnameVerifier(allHostsValid);
} catch (NoSuchAlgorithmException | KeyManagementException e) {
    e.printStackTrace();
}
```

チャンネルをしてファイルをダウンロードする

ファイルがすでにするはきされます。

```
String fileName      = "file.zip";                // name of the file
String urlToGetFrom  = "http://www.mywebsite.com/"; // URL to get it from
String pathToSaveTo  = "C:\\Users\\user\\";         // where to put it

//If the file already exists, it will be overwritten!

//Opening OutputStream to the destination file
try (ReadableByteChannel rbc =
    Channels.newChannel(new URL(urlToGetFrom + fileName).openStream())) {
    try (FileChannel channel =
        new FileOutputStream(pathToSaveTo + fileName).getChannel()); {
        channel.transferFrom(rbc, 0, Long.MAX_VALUE);
    }
    catch (FileNotFoundException e) { /* Output directory not found */ }
    catch (IOException e)           { /* File IO error */ }
}
catch (MalformedURLException e)    { /* URL is malformed */ }
catch (IOException e)              { /* IO error connecting to website */ }
```

ノート

- キャッチブロックをにしないでください
- エラーがしたは、リモートファイルがするかどうかをしてください
- これはブロックであり、きなファイルではがかかる

オンラインでネットワーキングをむ <https://riptutorial.com/ja/java/topic/149/ネットワーキング>

111: バイトコードの

Examples

バイトコードとはですか

バイトコードは、JVMでされるのです。これをするために、このHello Worldプログラムをしましょう。

```
public static void main(String[] args){
    System.out.println("Hello World");
}
```

これは、バイトコードにコンパイルされたときにわかります。

```
public static main([Ljava/lang/String; args)V
    getstatic java/lang/System out Ljava/io/PrintStream;
    ldc "Hello World"
    invokevirtual java/io/PrintStream print(Ljava/lang/String;)V
```

これのにあるはですか

getstatic - クラスのフィールドのをします。この、 *PrintStream*はシステムの "Out"になります。

ldc - をスタックにプッシュします。この、 "Hello World"

invokevirtual - スタックのロードされたのメソッドをびし、そのをスタックにします。メソッドのパラメータもスタックからされます。

まあ、もっといいことがありますか

255のオペコードがありますが、それらのすべてがまだされていません。のオペコードをすべてむテーブルがここにあります [Javaバイトコードリスト](#)。

バイトコードのきみ/はどうすればできますか

バイトコードのきみとにはのがあります。コンパイラの、ライブラリの、プログラムのがです。

くため

- [ジャスミン](#)

- クラカタウ

のため

- - ASM
 - Javassist
 - BCEL - Java 8+をサポートしていません
- ツール
 - バイトコード - ビューア
 - JBytedit
 - reJ - Java 8+をサポートしていません
 - JBE - Java 8+をサポートしていません

はバイトコードについてもっとびたいとっています

たぶん、バイトコードのためのなドキュメンテーションページがあります。このページでは、さまざまなライブラリとツールをしてバイトコードをするについてします。

ASMでjarファイルをする

まず、からのクラスをロードするがあります。このプロセスには3つのがあります

- loadClassesFile
- readJarJarFile、JarEntry、Map
- getNodebyte []

```
Map<String, ClassNode> loadClasses(File jarFile) throws IOException {
    Map<String, ClassNode> classes = new HashMap<String, ClassNode>();
    JarFile jar = new JarFile(jarFile);
    Stream<JarEntry> str = jar.stream();
    str.forEach(z -> readJar(jar, z, classes));
    jar.close();
    return classes;
}

Map<String, ClassNode> readJar(JarFile jar, JarEntry entry, Map<String, ClassNode> classes) {
    String name = entry.getName();
    try (InputStream jis = jar.getInputStream(entry)){
        if (name.endsWith(".class")) {
            byte[] bytes = IOUtils.toByteArray(jis);
            String cafebabe = String.format("%02X%02X%02X%02X", bytes[0], bytes[1], bytes[2],
bytes[3]);
            if (!cafebabe.toLowerCase().equals("cafebabe")) {
                // This class doesn't have a valid magic
                return classes;
            }
            try {
                ClassNode cn = getNode(bytes);
                classes.put(cn.name, cn);
            }
        }
    }
}
```

```

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
} catch (IOException e) {
    e.printStackTrace();
}
return classes;
}

ClassNode getNode(byte[] bytes) {
    ClassReader cr = new ClassReader(bytes);
    ClassNode cn = new ClassNode();
    try {
        cr.accept(cn, ClassReader.EXPAND_FRAMES);
    } catch (Exception e) {
        e.printStackTrace();
    }
    cr = null;
    return cn;
}

```

これらのメソッドをすると、jarファイルのみみとはマップのClassNodesをするなになります。このでは、Tree APIをして、jarのすべてのStringをのStringにきえます。

```

File jarFile = new File("sample.jar");
Map<String, ClassNode> nodes = loadClasses(jarFile);
// Iterate ClassNodes
for (ClassNode cn : nodes.values()){
    // Iterate methods in class
    for (MethodNode mn : cn.methods){
        // Iterate instructions in method
        for (AbstractInsnNode ain : mn.instructions.toArray()){
            // If the instruction is loading a constant value
            if (ain.getOpcode() == Opcodes.LDC){
                // Cast current instruction to Ldc
                // If the constant is a string then capitalize it.
                LdcInsnNode ldc = (LdcInsnNode) ain;
                if (ldc.cst instanceof String){
                    ldc.cst = ldc.cst.toString().toUpperCase();
                }
            }
        }
    }
}
}

```

すべてのClassNodeのがされたので、をするがあります。をしてをするには、いくつかのことをうがあります。

- ClassNodesをバイトにエクスポートする
- クラスのjarエントリ *Manifest.mf* / *jar*のそののバイナリリソースをバイトとしてロードする
- すべてのバイトをしいjarにする

ののから、3つのメソッドをします。

- processNodesMap <String、ClassNode> nodes

- loadNonClassesFile jarFile
- saveAsJarMap <String, byte []> outBytes, String fileName

```
Map<String, byte[]> out = process(nodes, new HashMap<String, MappedClass>());
out.putAll(loadNonClassEntries(jarFile));
saveAsJar(out, "sample-edit.jar");
```

される

```
static Map<String, byte[]> processNodes(Map<String, ClassNode> nodes, Map<String, MappedClass>
mappings) {
    Map<String, byte[]> out = new HashMap<String, byte[]>();
    // Iterate nodes and add them to the map of <Class names , Class bytes>
    // Using Compute_Frames ensures that stack-frames will be re-calculated automatically
    for (ClassNode cn : nodes.values()) {
        ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
        out.put(mappings.containsKey(cn.name) ? mappings.get(cn.name).getNewName() : cn.name,
cw.toByteArray());
    }
    return out;
}

static Map<String, byte[]> loadNonClasses(File jarFile) throws IOException {
    Map<String, byte[]> entries = new HashMap<String, byte[]>();
    ZipInputStream jis = new ZipInputStream(new FileInputStream(jarFile));
    ZipEntry entry;
    // Iterate all entries
    while ((entry = jis.getNextEntry()) != null) {
        try {
            String name = entry.getName();
            if (!name.endsWith(".class") && !entry.isDirectory()) {
                // Apache Commons - byte[] toByteArray(InputStream input)
                //
                // Add each entry to the map <Entry name , Entry bytes>
                byte[] bytes = IOUtils.toByteArray(jis);
                entries.put(name, bytes);
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            jis.closeEntry();
        }
    }
    jis.close();
    return entries;
}

static void saveAsJar(Map<String, byte[]> outBytes, String fileName) {
    try {
        // Create jar output stream
        JarOutputStream out = new JarOutputStream(new FileOutputStream(fileName));
        // For each entry in the map, save the bytes
        for (String entry : outBytes.keySet()) {
            // Append class names to class entries
            String ext = entry.contains(".") ? "" : ".class";
            out.putNextEntry(new ZipEntry(entry + ext));
            out.write(outBytes.get(entry));
            out.closeEntry();
        }
    }
```

```

        out.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

それでおしまい。すべてののは "sample-edit.jar"にされます。

ClassNodeをクラスとしてロードする

```

/**
 * Load a class by from a ClassNode
 *
 * @param cn
 *         ClassNode to load
 * @return
 */
public static Class<?> load(ClassNode cn) {
    ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
    return new ClassDefiner(ClassLoader.getSystemClassLoader()).get(cn.name.replace("/", "."),
        cw.toByteArray());
}

/**
 * Classloader that loads a class from bytes.
 */
static class ClassDefiner extends ClassLoader {
    public ClassDefiner(ClassLoader parent) {
        super(parent);
    }

    public Class<?> get(String name, byte[] bytes) {
        Class<?> c = defineClass(name, bytes, 0, bytes.length);
        resolveClass(c);
        return c;
    }
}

```

jar ファイルのクラスのをする

```

public static void main(String[] args) throws Exception {
    File jarFile = new File("Input.jar");
    Map<String, ClassNode> nodes = JarUtils.loadClasses(jarFile);

    Map<String, byte[]> out = JarUtils.loadNonClassEntries(jarFile);
    Map<String, String> mappings = new HashMap<String, String>();
    mappings.put("me/example/ExampleClass", "me/example/ExampleRenamed");
    out.putAll(process(nodes, mappings));
    JarUtils.saveAsJar(out, "Input-new.jar");
}

static Map<String, byte[]> process(Map<String, ClassNode> nodes, Map<String, String> mappings)
{
    Map<String, byte[]> out = new HashMap<String, byte[]>();
    Remapper mapper = new SimpleRemapper(mappings);
    for (ClassNode cn : nodes.values()) {
        ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
    }
}

```

```

        ClassVisitor remapper = new ClassRemapper(cw, mapper);
        cn.accept(remapper);
        out.put(mappings.containsKey(cn.name) ? mappings.get(cn.name) : cn.name,
            cw.toByteArray());
    }
    return out;
}

```

SimpleRemapperは、ASMライブラリののクラスです。ただし、クラスののみがです。フィールドとメソッドのをするは、Remapperクラスののをするがあります。

Javassist Basic

Javassistはバイトコードライブラリで、Javassistによってバイトコードにされ、にインストルメントされたクラス/メソッドにされるJavaコードをするバイトコードをできるようにします。

にクラス "com.my.to.be.instrumented.MyClass"をるのトランスフォーマーをし、メソッドのにログびしをします。

```

import java.lang.instrument.ClassFileTransformer;
import java.lang.instrument.IllegalClassFormatException;
import java.security.ProtectionDomain;
import javassist.ClassPool;
import javassist.CtClass;
import javassist.CtMethod;

public class DynamicTransformer implements ClassFileTransformer {

    public byte[] transform(ClassLoader loader, String className, Class classBeingRedefined,
        ProtectionDomain protectionDomain, byte[] classfileBuffer) throws
        IllegalClassFormatException {

        byte[] byteCode = classfileBuffer;

        // into the transformer will arrive every class loaded so we filter
        // to match only what we need
        if (className.equals("com/my/to/be/instrumented/MyClass")) {

            try {
                // retrieve default Javassist class pool
                ClassPool cp = ClassPool.getDefault();
                // get from the class pool our class with this qualified name
                CtClass cc = cp.get("com.my.to.be.instrumented.MyClass");
                // get all the methods of the retrieved class
                CtMethod[] methods = cc.getDeclaredMethods()
                for(CtMethod meth : methods) {
                    // The instrumentation code to be returned and injected
                    final StringBuffer buffer = new StringBuffer();
                    String name = meth.getName();
                    // just print into the buffer a log for example
                    buffer.append("System.out.println(\"Method \" + name + \" executed\");");
                    meth.insertBefore(buffer.toString())
                }
                // create the byteclode of the class
                byteCode = cc.toBytecode();
                // remove the CtClass from the ClassPool
                cc.detach();
            }
        }
    }
}

```

```

        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    return byteCode;
}
}

```

このトランスフォーマーをするにはロードにJVMがクラスのメソッドをびすように、このインストーラにエージェントをするがあります。

```

import java.lang.instrument.Instrumentation;

public class EasyAgent {

    public static void premain(String agentArgs, Instrumentation inst) {

        // registers the transformer
        inst.addTransformer(new DynamicTransformer());
    }
}

```

のをするのステップは、このエージェントクラスをにJVMマシンのにすることです。にうもなは、シェルコマンドにオプションですることです。

```

java -javaagent:myAgent.jar MyJavaApplication

```

たちがることができるように、エージェント/トランスフォーマープロジェクトは、にたちのされたコードをするために "com.my.to.be.instrumented.MyClass" というのクラスをむはずの MyJavaApplication というのアプリケーションのにjarとしてされます。

オンラインでバイトコードのをむ <https://riptutorial.com/ja/java/topic/3747/バイトコードの>

112: パッケージ

き

javaのパッケージは、クラスとインタフェースをグループするためにされます。ななクラスがある、これによりはをけることができます。このパッケージをすると、なるパッケージにじのクラス/インタフェースをできます。パッケージをすることで、もう1つのクラスでそのピースをインポートすることができます。 > 1.java.util> 2.java.lang> 3.java.ioのようなjavaには、くのパッケージがみま れています。のユーザーパッケージをできます。

パッケージはアクセスをします。

packageは、ソースコードの1でなければなりません。1つのソースファイルには1つのパッケージしかしません。

パッケージのけをりて、なるモジュールのをすることができます。

Examples

パッケージをしてじのクラスをする

のTest.class

```
package foo.bar

public class Test {

}
```

のパッケージのTest.class

```
package foo.bar.baz

public class Test {

}
```

2つのクラスがなるパッケージにするため、はうまくいきます。

パッケージの

Javaでは、アクセスをしないと、のデフォルトスコープはパッケージレベルになります。つまり、これらののがにであるかのように、クラスがじパッケージののクラスのにアクセスすることができます。

```

package foo.bar

public class ExampleClass {
    double exampleNumber;
    String exampleString;

    public ExampleClass() {
        exampleNumber = 3;
        exampleString = "Test String";
    }
    //No getters or setters
}

package foo.bar

public class AnotherClass {
    ExampleClass clazz = new ExampleClass();

    System.out.println("Example Number: " + clazz.exampleNumber);
    //Prints Example Number: 3
    System.out.println("Example String: " + clazz.exampleString);
    //Prints Example String: Test String
}

```

このメソッドは、のパッケージのクラスではありません。

```

package baz.foo

public class ThisShouldNotWork {
    ExampleClass clazz = new ExampleClass();

    System.out.println("Example Number: " + clazz.exampleNumber);
    //Throws an exception
    System.out.println("Example String: " + clazz.exampleString);
    //Throws an exception
}

```

オンラインでパッケージをむ <https://riptutorial.com/ja/java/topic/8273/パッケージ>

113: ハッシュ

き

Hashtableは、Mapインタフェースをし、DictionaryクラスをするJavaコレクションのクラスです。

ユニークなとそののみがまれています

Examples

ハッシュ

```
import java.util.*;
public class HashtableDemo {
    public static void main(String args[]) {
        // create and populate hash table
        Hashtable<Integer, String> map = new Hashtable<Integer, String>();
        map.put(101, "C Language");
        map.put(102, "Domain");
        map.put(104, "Databases");
        System.out.println("Values before remove: " + map);
        // Remove value for key 102
        map.remove(102);
        System.out.println("Values after remove: " + map);
    }
}
```

オンラインでハッシュをむ <https://riptutorial.com/ja/java/topic/10709/ハッシュ>

114: ビット

- C/C++とはなり、Javaはとなるマシンハードウェアにしてにエンディアンニュートラルです。デフォルトでは、エンディアンのがきくなることはありません。あなたはにあなたがむをしなければなりません。
- `byte`はきで、は-128+ 127です。バイトをなしのにするには、のように0xFFでマスクします
(`b & 0xFF`)。

Examples

ビットフラグメントとしてののパッキング/アンパック

メモリのパフォーマンスでは、のをのプリミティブにするのがです。これは、さまざまなのをのにするにです。

たとえば、**RGB**のカラーコードなどの3バイトをのintにパックすることができます。

のパッキング

```
// Raw bytes as input
byte[] b = {(byte)0x65, (byte)0xFF, (byte)0x31};

// Packed in big endian: x == 0x65FF31
int x = (b[0] & 0xFF) << 16 // Red
      | (b[1] & 0xFF) << 8  // Green
      | (b[2] & 0xFF) << 0; // Blue

// Packed in little endian: y == 0x31FF65
int y = (b[0] & 0xFF) << 0
      | (b[1] & 0xFF) << 8
      | (b[2] & 0xFF) << 16;
```

のアンパック

```
// Raw int32 as input
int x = 0x31FF65;

// Unpacked in big endian: {0x65, 0xFF, 0x31}
byte[] c = {
    (byte)(x >> 16),
    (byte)(x >> 8),
    (byte)(x & 0xFF)
};

// Unpacked in little endian: {0x31, 0xFF, 0x65}
byte[] d = {
    (byte)(x & 0xFF),
    (byte)(x >> 8),
    (byte)(x >> 16)
};
```


々のビットのチェック、、クリア、トグル。いビットマスクとしてする

プリミティブのビット_nをしたいとすると、_i byte、short、char、int、またはlong

```
(i & 1 << n) != 0 // checks bit 'n'
i |= 1 << n;      // sets bit 'n' to 1
i &= ~(1 << n);   // sets bit 'n' to 0
i ^= 1 << n;      // toggles the value of bit 'n'
```

long / int / short / byteをビットマスクとしてう

```
public class BitMaskExample {
    private static final long FIRST_BIT = 1L << 0;
    private static final long SECOND_BIT = 1L << 1;
    private static final long THIRD_BIT = 1L << 2;
    private static final long FOURTH_BIT = 1L << 3;
    private static final long FIFTH_BIT = 1L << 4;
    private static final long BIT_55 = 1L << 54;

    public static void main(String[] args) {
        checkBitMask(FIRST_BIT | THIRD_BIT | FIFTH_BIT | BIT_55);
    }

    private static void checkBitMask(long bitmask) {
        System.out.println("FIRST_BIT: " + ((bitmask & FIRST_BIT) != 0));
        System.out.println("SECOND_BIT: " + ((bitmask & SECOND_BIT) != 0));
        System.out.println("THIRD_BIT: " + ((bitmask & THIRD_BIT) != 0));
        System.out.println("FOURTH_BIT: " + ((bitmask & FOURTH_BIT) != 0));
        System.out.println("FIFTH_BIT: " + ((bitmask & FIFTH_BIT) != 0));
        System.out.println("BIT_55: " + ((bitmask & BIT_55) != 0));
    }
}
```

プリント

```
FIRST_BIT: true
SECOND_BIT: false
THIRD_BIT: true
FOURTH_BIT: false
FIFTH_BIT: true
BIT_55: true
```

これはcheckBitMaskパラメータとしてされたマスクにしますFIRST_BIT | THIRD_BIT | FIFTH_BIT | BIT_55。

2のをする

の 2^2 を_nをするために、_nをにできるビットシフトをすることができます。

はにはのとおりで。

```
int pow2 = 1<<n;
```

```
int twoExp4 = 1<<4; //2^4
int twoExp5 = 1<<5; //2^5
int twoExp6 = 1<<6; //2^6
...
int twoExp31 = 1<<31; //2^31
```

これは、16または10をするわりに、2のがされることがらかになるはずのをするににです。

```
int twoExp4 = 0x10; //hexadecimal
int twoExp5 = 0x20; //hexadecimal
int twoExp6 = 64; //decimal
...
int twoExp31 = -2147483648; //is that a power of 2?
```

2のintのべきをするなは、

```
int pow2(int exp){
    return 1<<exp;
}
```

が2のであるかどうかをする

x が2のである、1ビットのみがされ、 $x-1$ はそれにされるすべてのビットをする。えは、 4 が 100 あり、 3 が2で 011 であり、のをたす。ゼロは2ではなく、にチェックするがあります。

```
boolean isPowerOfTwo(int x)
{
    return (x != 0) && ((x & (x - 1)) == 0);
}
```

シフトとシフトの

READ、**WRITE**、**EXECUTE**という3のパーミッションがあるとしましょう。のは0から7までです4ビットのシステムをしましょう

RESOURCE = READ WRITE EXECUTE12ビット

RESOURCE = 0100 0110 0101 = 4 6 512ビット

12ビットでした12ビットのパーミッションをどのようにできますか

0100 0110 0101

0000 0000 0111

0000 0000 0101 = 5

したがって、これが**RESOURCE**の **EXECUTE**パーミッションをするです。さて、リソースの **READ**をしたいのですが

0100 0110 0101

0111 0000 0000

0100 0000 0000 = 1024

あなたはおそらくこれをしていきますが、は1024にこります。私たちはそのリソースにして
READしかられません。しないでください。そういうわけで、シフトがあります。たちがると、
READはのより8ビットれているので、のにREADをえるシフトをすると、もしたちがすれば

0100 0000 0000 >> 8 => 0000 0000 0100ので0にきえられているため、をにしないは
、なしのシフトをしてください

には4つの**READ**があります。

、たとえば、 リソースにして**READ**、 **WRITE**、 **EXECUTE**パーミッションがえられています
、このリソースにしてパーミッションをえるにはどうすればよいですか

にバイナリパーミッションのをてみましょう。 まだ4ビットのシステムをしています

READ = 0001

WRITE = 0100

EXECUTE = 0110

あなたがにたちがやることをえているなら、

READ | WRITE | EXECUTE、あなたはちょっといいですがにはありません。してください、もしたちが
がREAD | きみ | する

0001 | 0100 | 0110 => 0111

しかし、パーミッションはにはこのでは0001 0100 0110

ですから、これをうには、 **READ**が8ビットれ、 **WRITE**が4ビットれ、 **PERMISSIONS**がにされ
ていることがわかります。 **RESOURCE**のためにされているシステムはには12ビットですこの
で。これは、なるシステムでなることができますことができます。

READ << 8 | WRITE << 4 |

0000 0000 0001 << 8 READ

0001 0000 0000 8ビットシフト

0000 0000 0100 << 4 WRITE

0000 0100 0000 4ビットシフト

0000 0000 0001

、のシフトのをすると、それはのようになります。

0001 0000 0000READ

0000 0100 0000きみ

0000 0000 0001

0001 0100 0001

java.util.BitSet クラス

1.7、シンプルでしやすいビットストレージとインタフェースをする[java.util.BitSet](#)クラスがあります。

```
final BitSet bitSet = new BitSet(8); // by default all bits are unset

IntStream.range(0, 8).filter(i -> i % 2 == 0).forEach(bitSet::set); // {0, 2, 4, 6}

bitSet.set(3); // {0, 2, 3, 4, 6}

bitSet.set(3, false); // {0, 2, 4, 6}

final boolean b = bitSet.get(3); // b = false

bitSet.flip(6); // {0, 2, 4}

bitSet.set(100); // {0, 2, 4, 100} - expands automatically
```

BitSetはClonableとSerializableしており、ワードのすべてのビットはにされるlong[] wordsフィールドにされます。

また、のセットのをサポートand、or、xor、andNot

```
bitSet.and(new BitSet(8));
bitSet.or(new BitSet(8));
bitSet.xor(new BitSet(8));
bitSet.andNot(new BitSet(8));
```

きシフトとなしシフト

Javaでは、すべてのプリミティブがされています。たとえば、intはに $[-2^{31} - 1, 2^{31}]$ のをし、のビットはのは-1、のは0のにします。

シフト>>および<<はきです。らはのをつでしよう。

しかし、プログラマは、なしのをするためにをするのがです。intの、を $[0, 2^{32} - 1]$ にシフトし、きintのの2のをつことをします。

これらのパワーユーザーの、のないのビット。そのため、Javaでは、シフトをしてシフト>>>しました。

```

        initial value:          4 (          100)
    signed left-shift: 4 << 1      8 (        1000)
    signed right-shift: 4 >> 1      2 (          10)
    unsigned right-shift: 4 >>> 1  2 (          10)
        initial value:        -4 ( 1111111111111111111111111111100)
    signed left-shift: -4 << 1     -8 ( 111111111111111111111111111000)
    signed right-shift: -4 >> 1     -2 ( 111111111111111111111111111110)
    unsigned right-shift: -4 >>> 1 2147483646 ( 111111111111111111111111111110)

```

なぜ<<<はありませんか

これは、シフトのされたにします。にいているをめるので、サインのビットについてはをすことはありません。として、2つのなるのはない。

よりしいについては、このをしてください。

オンラインでビットをむ <https://riptutorial.com/ja/java/topic/1177/ビット>

115: ファイルI/O

き

Java I/O Input and Outputは、をしてをするためにされます。JavaはI/Oをするためにストリームというをしています。java.ioパッケージには、になすべてのクラスがまれています。ファイルのは、Java I/O APIによってJavaでもわれます。

Examples

すべてのバイトをバイト[]にみむ

Java 7はになファイルクラスを

Java SE 7

```
import java.nio.file.Files;
import java.nio.file.Paths;
import java.nio.file.Path;

Path path = Paths.get("path/to/file");

try {
    byte[] data = Files.readAllBytes(path);
} catch (IOException e) {
    e.printStackTrace();
}
```

ファイルからをみむ

```
import java.awt.Image;
import javax.imageio.ImageIO;

...

try {
    Image img = ImageIO.read(new File("~/Desktop/cat.png"));
} catch (IOException e) {
    e.printStackTrace();
}
```

バイト[]をファイルにきむ

Java SE 7

```
byte[] bytes = { 0x48, 0x65, 0x6c, 0x6c, 0x6f };

try (FileOutputStream stream = new FileOutputStream("Hello world.txt")) {
    stream.write(bytes);
}
```

```

} catch (IOException ioe) {
    // Handle I/O Exception
    ioe.printStackTrace();
}

```

Java SE 7

```

byte[] bytes = { 0x48, 0x65, 0x6c, 0x6c, 0x6f };

FileOutputStream stream = null;
try {
    stream = new FileOutputStream("Hello world.txt");
    stream.write(bytes);
} catch (IOException ioe) {
    // Handle I/O Exception
    ioe.printStackTrace();
} finally {
    if (stream != null) {
        try {
            stream.close();
        } catch (IOException ignored) {}
    }
}

```

ほとんどのjava.ioファイルAPIは、として `String` と `File` のをくれます。

```

File file = new File("Hello world.txt");
FileOutputStream stream = new FileOutputStream(file);

```

Stream vs Writer / Reader API

ストリームはバイナリコンテンツへのものなアクセスをするので、 `InputStream` / `OutputStream` はに `int` と `byte` し `byte` 。

```

// Read a single byte from the stream
int b = inputStream.read();
if (b >= 0) { // A negative value represents the end of the stream, normal values are in the
    range 0 - 255
    // Write the byte to another stream
    outputStream.write(b);
}

// Read a chunk
byte[] data = new byte[1024];
int nBytesRead = inputStream.read(data);
if (nBytesRead >= 0) { // A negative value represents end of stream
    // Write the chunk to another stream
    outputStream.write(data, 0, nBytesRead);
}

```

いくつかのがあります。おそらく `PrintStream` は、「さまざまなデータのをにする」をしています。これにより、 `System.out.println()` などのメソッドをして、バイナリ `InputStream` とテキストのとして `System.out` をできます。

また、のストリームは、Javaオブジェクト `DataOutputStream` やネイティブ `DataOutputStream / DataInputStream` などなどのレベルのコンテンツへのインタフェースとします。

`Writer` クラスと `Reader` クラスでは、JavaはなストリームのAPIもしています。ほとんどのアプリケーションはこれらのをストリームにしていますが、ストリームAPIはバイナリコンテンツのメソッドをしません。

```
// This example uses the platform's default charset, see below
// for a better implementation.
```

```
Writer writer = new OutputStreamWriter(System.out);
writer.write("Hello world!");
```

```
Reader reader = new InputStreamReader(System.in);
char singleCharacter = reader.read();
```

をバイナリデータにエンコードするがあるときはいつでもたとえば `InputStreamWriter / OutputStreamWriter` クラスをするとき、プラットフォームのデフォルトの `charset` にしたくないは、`charset` をするがあります。かなは、すべてのJavaプラットフォームでサポートされているUTF-8などのUnicodeのエンコーディングをしてください。したがって、`FileWriter` や `FileReader` ようなクラスは、にデフォルトのプラットフォームの `charset` をするクラスかられているがあります。ストリームをしてファイルにアクセスするよりいのはのとおりです。

```
Charset myCharset = StandardCharsets.UTF_8;
```

```
Writer writer = new OutputStreamWriter( new FileOutputStream("test.txt"), myCharset );
writer.write('Ä');
writer.flush();
writer.close();
```

```
Reader reader = new InputStreamReader( new FileInputStream("test.txt"), myCharset );
char someUnicodeCharacter = reader.read();
reader.close();
```

もにされる `Reader` の1つは `BufferedReader` これはのリーダーからのテキストをみるメソッドをし、ごとにストリームをみむもなです

```
// Read from baseReader, one line at a time
BufferedReader reader = new BufferedReader( baseReader );
String line;
while((line = reader.readLine()) != null) {
    // Remember: System.out is a stream, not a writer!
    System.out.println(line);
}
```

にファイルをむ

```
File f = new File(path);
String content = new Scanner(f).useDelimiter("\\Z").next();
```

`\Z`はEOFファイルのわりです。りとしてすると、スキャナはEOFフラグにするまでりつぶしをみ

みます。

スキャナーでファイルをむ

でファイルを読みむ

```
public class Main {

    public static void main(String[] args) {
        try {
            Scanner scanner = new Scanner(new File("example.txt"));
            while(scanner.hasNextLine())
            {
                String line = scanner.nextLine();
                //do stuff
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

```
public class Main {

    public static void main(String[] args) {
        try {
            Scanner scanner = new Scanner(new File("example.txt"));
            while(scanner.hasNext())
            {
                String line = scanner.next();
                //do stuff
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

また、`scanner.useDelimiter`メソッドをしてりをすることもできます

ディレクトリをフィルタリングし、ファイルでフィルタリングする

```
public void iterateAndFilter() throws IOException {
    Path dir = Paths.get("C:/foo/bar");
    PathMatcher imageFileMatcher =
        FileSystems.getDefault().getPathMatcher(
            "regex:.*(?i:jpg|jpeg|png|gif|bmp|jpe|jfif)");

    try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir,
        entry -> imageFileMatcher.matches(entry.getFileName()))) {

        for (Path path : stream) {
            System.out.println(path.getFileName());
        }
    }
}
```

java.io.FileからJava 7へのNIOjava.nio.file.Path

これらでは、Java 7のNIOがであることをにっており、`java.io.File`してコードを書くことにしていることをとしています。これらをして、のためのNIOのドキュメントをすばやくつけることができます。

Java 7のNIOには、[メモリマップされたファイル](#)や、[FileSystem](#)をつてZIPファイルやJARファイルをくなどのがあります。これらは、られたのなユースケースをカバーするだけです。

なルールとして、`java.io.File`インスタンス・メソッドをしてファイル・システムのみり/きみをする、`java.nio.file.Files`のメソッドとしてされます。

パスをす

```
// -> IO
File file = new File("io.txt");

// -> NIO
Path path = Paths.get("nio.txt");
```

のパスにするパス

```
// Forward slashes can be used in place of backslashes even on a Windows operating system
// -> IO
File folder = new File("C:/");
File fileInFolder = new File(folder, "io.txt");

// -> NIO
Path directory = Paths.get("C:/");
Path pathInDirectory = directory.resolve("nio.txt");
```

ライブラリとののためのファイルのパスの

```
// -> IO to NIO
Path pathFromFile = new File("io.txt").toPath();

// -> NIO to IO
File fileFromPath = Paths.get("nio.txt").toFile();
```

ファイルがするかどうかをチェックし、するはします

```
// -> IO
if (file.exists()) {
    boolean deleted = file.delete();
    if (!deleted) {
        throw new IOException("Unable to delete file");
    }
}

// -> NIO
Files.deleteIfExists(path);
```

OutputStreamでファイルにきむ

`FileChannel`、`Files.write(Path path, byte[] bytes, OpenOption... options)`など、さまざまなパフォーマンスとメモリの、みやすさでNIOをしてファイルからみきするはいくつかあり

`Files.write(Path path, byte[] bytes, OpenOption... options)`。..このでは、`OutputStream`のみを`OutputStream`ますが、メモリマップファイルと`java.nio.file.Files`なさまざまなメソッドについてぶことをくおめします。

```
List<String> lines = Arrays.asList(
    String.valueOf(Calendar.getInstance().getTimeInMillis()),
    "line one",
    "line two");

// -> IO
if (file.exists()) {
    // Note: Not atomic
    throw new IOException("File already exists");
}
try (FileOutputStream outputStream = new FileOutputStream(file)) {
    for (String line : lines) {
        outputStream.write((line + System.lineSeparator()).getBytes(StandardCharsets.UTF_8));
    }
}

// -> NIO
try (OutputStream outputStream = Files.newOutputStream(path, StandardOpenOption.CREATE_NEW)) {
    for (String line : lines) {
        outputStream.write((line + System.lineSeparator()).getBytes(StandardCharsets.UTF_8));
    }
}
```

フォルダのファイルをりしする

```
// -> IO
for (File selectedFile : folder.listFiles()) {
    // Note: Depending on the number of files in the directory folder.listFiles() may take a
    long time to return
    System.out.println((selectedFile.isDirectory() ? "d" : "f") + " " +
        selectedFile.getAbsolutePath());
}
```

```
// -> NIO
Files.walkFileTree(directory, EnumSet.noneOf(FileVisitOption.class), 1, new
SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult preVisitDirectory(Path selectedPath, BasicFileAttributes attrs)
throws IOException {
        System.out.println("d " + selectedPath.toAbsolutePath());
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFile(Path selectedPath, BasicFileAttributes attrs) throws
IOException {
        System.out.println("f " + selectedPath.toAbsolutePath());
        return FileVisitResult.CONTINUE;
    }
});
```

フォルダの

```
// -> IO
recurseFolder(folder);

// -> NIO
// Note: Symbolic links are NOT followed unless explicitly passed as an argument to
Files.walkFileTree
Files.walkFileTree(directory, new SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs) throws
IOException {
        System.out.println("d " + selectedPath.toAbsolutePath());
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFile(Path selectedPath, BasicFileAttributes attrs) throws
IOException {
        System.out.println("f " + selectedPath.toAbsolutePath());
        return FileVisitResult.CONTINUE;
    }
});

private static void recurseFolder(File folder) {
    for (File selectedFile : folder.listFiles()) {
        System.out.println((selectedFile.isDirectory() ? "d" : "f") + " " +
selectedFile.getAbsolutePath());
        if (selectedFile.isDirectory()) {
            // Note: Symbolic links are followed
            recurseFolder(selectedFile);
        }
    }
}
```

FileInputStream / FileOutputStreamをしたファイルのみき

ファイルにきむtest.txt

```
String filepath = "C:\\test.txt";
FileOutputStream fos = null;
try {
    fos = new FileOutputStream(filepath);
    byte[] buffer = "This will be written in test.txt".getBytes();
    fos.write(buffer, 0, buffer.length);
    fos.close();
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
} finally{
    if(fos != null)
        fos.close();
}
```

ファイルtest.txtからのみり

```
String filepath = "C:\\test.txt";
FileInputStream fis = null;
try {
    fis = new FileInputStream(filepath);
    int length = (int) new File(filepath).length();
    byte[] buffer = new byte[length];
    fis.read(buffer, 0, length);
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
} finally{
    if(fis != null)
        fis.close();
}
```

Java 1.7、[try-with-resources](#)ステートメントがされたことで、みきのはずっとになりました。

ファイルにきむtest.txt

```
String filepath = "C:\\test.txt";
try (FileOutputStream fos = new FileOutputStream(filepath)){
    byte[] buffer = "This will be written in test.txt".getBytes();
    fos.write(buffer, 0, buffer.length);
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

ファイルtest.txtからのみり

```
String filepath = "C:\\test.txt";
try (FileInputStream fis = new FileInputStream(filepath)){
    int length = (int) new File(filepath).length();
    byte[] buffer = new byte[length];
```

```

        fis.read(buffer, 0, length);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

バイナリファイルからのみみ

Javaのすべてのバージョンで、のコードをしてバイナリファイルをむことができます。

Java SE 1.4

```

File file = new File("path_to_the_file");
byte[] data = new byte[(int) file.length()];
DataInputStream stream = new DataInputStream(new FileInputStream(file));
stream.readFully(data);
stream.close();

```

Java 7をしているは、 `nio API` をするよりながあり `nio API` 。

Java SE 7

```

Path path = Paths.get("path_to_the_file");
byte [] data = Files.readAllBytes(path);

```

ロッキング

ファイルは、 `streams` および `readers` からできる `FileChannel API` をしてロックできます

`streams`

//ファイルストリームをきます `FileInputStream ios = new FileInputStream(filename;` //ファイルストリームをきます。

```

// get underlying channel
FileChannel channel = ios.getChannel();

/*
 * try to lock the file. true means whether the lock is shared or not i.e. multiple
processes can acquire a
 * shared lock (for reading only) Using false with readable channel only will generate an
exception. You should
 * use a writable channel (taken from FileOutputStream) when using false. tryLock will
always return immediately
 */
FileLock lock = channel.tryLock(0, Long.MAX_VALUE, true);

if (lock == null) {
    System.out.println("Unable to acquire lock");
} else {
    System.out.println("Lock acquired successfully");
}

```

```
// you can also use blocking call which will block until a lock is acquired.
channel.lock();

// Once you have completed desired operations of file. release the lock
if (lock != null) {
    lock.release();
}

// close the file stream afterwards
// Example with reader
RandomAccessFile randomAccessFile = new RandomAccessFile(filename, "rw");
FileChannel channel = randomAccessFile.getChannel();
//repeat the same steps as above but now you can use shared as true or false as the
channel is in read write mode
```

InputStreamとOutputStreamをしたファイルのコピー

ループをして、ソースからデータシンクにデータをコピーできます。このでは、InputStreamからデータをみむとに、OutputStreamにきんでいます。みきがわると、リソースをじるがあります。

```
public void copy(InputStream source, OutputStream destination) throws IOException {
    try {
        int c;
        while ((c = source.read()) != -1) {
            destination.write(c);
        }
    } finally {
        if (source != null) {
            source.close();
        }
        if (destination != null) {
            destination.close();
        }
    }
}
```

チャネルとバッファをしたファイルのみみ

Channelは、データをみきするためにBufferをします。バッファはサイズのコンテナで、に1ブロックのデータをきむことができます。ChannelはストリームベースのI/Oよりもかなりです。

Channelをしてファイルからデータをみむには、のをするがあります。

1. FileInputStreamインスタンスがです。FileInputStreamは、チャネルをすgetChannel()というメソッドがあります。
2. FileInputStreamのgetChannel()メソッドをびし、Channelをします。
3. ByteBufferをします。ByteBufferはサイズのバイトのコンテナです。
4. Channelにはreadメソッドがあり、このreadメソッドのとしてByteBufferをするがあります。ByteBufferには、みりムードときみムードという2つのモードがあります。flip()メソッドびしをしてモードをできます。バッファには、、およびがあります。サイズのバッファがされると、バッファのサイズとはじサイズになり、はゼロからまります。バッファにデータがきまれている、そのは々にします。モードのは、をすることをします。バッファのから

データを見るには、を0にするがあります。flipメソッドはをします。

5. Channel readメソッドをびすと、データをしてバッファがいっぱいになります。

6. ByteBuffer からデータを見るがあるは、バッファをしてそのモードをきみモードにしてから、バッファからデータを読み取るがあります。

7. みるデータがなくなった、channelのread() メソッドは0または-1をします。

```
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;

public class FileChannelRead {

    public static void main(String[] args) {

        File inputFile = new File("hello.txt");

        if (!inputFile.exists()) {
            System.out.println("The input file doesn't exist.");
            return;
        }

        try {
            FileInputStream fis = new FileInputStream(inputFile);
            FileChannel fileChannel = fis.getChannel();
            ByteBuffer buffer = ByteBuffer.allocate(1024);

            while (fileChannel.read(buffer) > 0) {
                buffer.flip();
                while (buffer.hasRemaining()) {
                    byte b = buffer.get();
                    System.out.print((char) b);
                }
                buffer.clear();
            }

            fileChannel.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

チャンネルをしたファイルのコピー

たちは、Channel をってファイルのをよりくコピーすることができます。これをうには、FileChannel transferTo() メソッドをできます。

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.channels.FileChannel;

public class FileCopier {
```



```

public static void main(String[] args) {
    File sourceFile = new File("hello.txt");
    File sinkFile = new File("hello2.txt");
    copy(sourceFile, sinkFile);
}

public static void copy(File sourceFile, File destFile) {
    if (!sourceFile.exists() || !destFile.exists()) {
        System.out.println("Source or destination file doesn't exist");
        return;
    }

    try (FileChannel srcChannel = new FileInputStream(sourceFile).getChannel();
        FileChannel sinkChannel = new FileOutputStream(destFile).getChannel()) {

        srcChannel.transferTo(0, srcChannel.size(), sinkChannel);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

BufferedInputStreamをしたファイルのみみ

BufferedInputStreamをしてファイルをみむのは、となるストリームからみられたバイトをするためのバッファをするため、FileInputStreamよりもにです。

```

import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;

public class FileReadingDemo {

    public static void main(String[] args) {
        String source = "hello.txt";

        try (BufferedInputStream bis = new BufferedInputStream(new FileInputStream(source))) {
            byte data;
            while ((data = (byte) bis.read()) != -1) {
                System.out.println((char) data);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

チャネルとバッファをしてファイルをきむ

Channelをしてファイルにデータをきむには、のがあります。

1. まず、FileOutputStreamオブジェクトをするがあります

2. `FileOutputStream` から `getChannel()` メソッドをびす `FileChannel` をします。
3. `ByteBuffer` をし、データでめます
4. に、 `ByteBuffer` の `flip()` メソッドをびして、それを `FileChannel write()` メソッドのとしてすがあります
5. がわったら、リソースをじるがあります

```
import java.io.*;
import java.nio.*;
public class FileChannelWrite {

    public static void main(String[] args) {

        File outputFile = new File("hello.txt");
        String text = "I love Bangladesh.";

        try {
            FileOutputStream fos = new FileOutputStream(outputFile);
            FileChannel fileChannel = fos.getChannel();
            byte[] bytes = text.getBytes();
            ByteBuffer buffer = ByteBuffer.wrap(bytes);
            fileChannel.write(buffer);
            fileChannel.close();
        } catch (java.io.IOException e) {
            e.printStackTrace();
        }
    }
}
```

PrintStream をしてファイルをきむ

`PrintStream` クラスをしてファイルをきむことができます。のデータのをできるいくつかのメソッドがあります。 `println()` メソッドはしいをします。がしたら、 `PrintStream` をフラッシュしなければなりません。

```
import java.io.FileNotFoundException;
import java.io.PrintStream;
import java.time.LocalDate;

public class FileWritingDemo {
    public static void main(String[] args) {
        String destination = "file1.txt";

        try(PrintStream ps = new PrintStream(destination)){
            ps.println("Stackoverflow documentation seems fun.");
            ps.println();
            ps.println("I love Java!");
            ps.printf("Today is: %1$tm/%1$td/%1$tY", LocalDate.now());

            ps.flush();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

ディレクトリのサブディレクトリをりしする

```
public void iterate(final String dirPath) throws IOException {
    final DirectoryStream<Path> paths = Files.newDirectoryStream(Paths.get(dirPath));
    for (final Path path : paths) {
        if (Files.isDirectory(path)) {
            System.out.println(path.getFileName());
        }
    }
}
```

ディレクトリの

`File` インスタンスからしいディレクトリをするには、`makedirs()` または `mkdir()` 2つののいずれかを
するがあります。

- `mkdir()` - このパスでされたディレクトリをします。 [ソース](#)
- `makedirs()` - このパスでされたディレクトリをします。なもしないディレクトリもめてします
。このがした、なディレクトリのいくつかをできたがあります。 [ソース](#)

`createNewFile()` はファイルのみしいディレクトリをしません。

```
File singleDir = new File("C:/Users/SomeUser/Desktop/A New Folder/");

File multiDir = new File("C:/Users/SomeUser/Desktop/A New Folder 2/Another Folder/");

// assume that neither "A New Folder" or "A New Folder 2" exist

singleDir.createNewFile(); // will make a new file called "A New Folder.file"
singleDir.mkdir(); // will make the directory
singleDir.mkdirs(); // will make the directory

multiDir.createNewFile(); // will throw a IOException
multiDir.mkdir(); // will not work
multiDir.mkdirs(); // will make the directory
```

エラーのブロックまたはリダイレクト

`System.err` てされたのライブラリは、ましくないを `System.out` または `System.err` ストリームに
`System.err` ます。されるは、よりいライブラリをつけるかオープンソースの、をし、にパッチをす
ることです。

のができないは、ストリームのリダイレクトをするがあります。

コマンドラインでのリダイレクト

UNIX、Linux または MacOSX システムでは、> リダイレクトをしてシェルからうことができます。
えは

```
$ java -jar app.jar arg1 arg2 > /dev/null 2>&1
```

```
$ java -jar app.jar arg1 arg2 > out.log 2> error.log
```

のものは、とエラーを `/dev/null` にリダイレクトします。これにより、それらのストリームにきまれたものはすべてされます。のは `"out.log"` に、エラーは `"error.log"` にリダイレクトされます。

リダイレクトのについては、しているコマンドシェルのドキュメントをしてください。のアドバイスはWindowsにされます。

また、Javaアプリケーションをするラッパー・スクリプトまたはバッチ・ファイルでリダイレクションをすることもできます。

Javaアプリケーションのリダイレクション

`System.setOut()` および `System.setErr()` をしてJavaアプリケーションでストリームをリダイレクトすることもできます。たとえば、のスニペットは、とエラーを2つのログファイルにリダイレクトします。

```
System.setOut(new PrintStream(new FileOutputStream(new File("out.log"))));
System.setErr(new PrintStream(new FileOutputStream(new File("err.log"))));
```

をにしたいは、なファイルに「きむ」ストリームをすることができます。これは、UNIXでは `/dev/null` へのきみとにです。

```
System.setOut(new PrintStream(new FileOutputStream(new FileDescriptor())));
System.setErr(new PrintStream(new FileOutputStream(new FileDescriptor())));
```

`setOut` と `setErr` いにしてください。

1. リダイレクションは、JVMにします。
2. これをうことで、コマンドラインからストリームをリダイレクトするユーザーのをうことになります。

ZIP ファイルのへのアクセス

Java 7のFileSystem APIは、のファイルシステムでのとじて、Java NIOファイルAPIをしてZipファイルのみみとをうことができます。

FileSystemはににクローズされるべきリソースなので、try-with-resourcesブロックをするがあります。

のファイルからのみみ

```
Path pathToZip = Paths.get("path/to/file.zip");
try(FileSystem zipFs = FileSystems.newFileSystem(pathToZip, null)) {
    Path root = zipFs.getPath("/");
    ... //access the content of the zip file same as ordinary files
} catch(IOException ex) {
```

```
ex.printStackTrace();
}
```

しいファイルをする

```
Map<String, String> env = new HashMap<>();
env.put("create", "true"); //required for creating a new zip file
env.put("encoding", "UTF-8"); //optional: default is UTF-8
URI uri = URI.create("jar:file:/path/to/file.zip");
try (FileSystem zipfs = FileSystems.newFileSystem(uri, env)) {
    Path newFile = zipfs.getPath("/newFile.txt");
    //writing to file
    Files.write(newFile, "Hello world".getBytes());
} catch(IOException ex) {
    ex.printStackTrace();
}
```

オンラインでファイルI/Oをむ <https://riptutorial.com/ja/java/topic/93/ファイルi---o>

116: フォーク/フレームワークによるプログラミング

Examples

Javaでフォーク/タスク

Javaのfork / joinフレームワークは、より簡単にしているにすぎません。フォーク/ジョインののとは異なります。

- をのにする
- それぞれのをいにく
- サブソリューションを1つのソリューションにまとめます

ForkJoinTaskは、このようなをするインタフェースです。は、インタフェースをするのではなく、そのは**RecursiveTask** の1つをサブクラスすることがにされています。

ここでは、10のバッチサイズになるまでのをします。

```
import java.util.List;
import java.util.concurrent.RecursiveTask;

public class SummingTask extends RecursiveTask<Integer> {
    private static final int MAX_BATCH_SIZE = 10;

    private final List<Integer> numbers;
    private final int minInclusive, maxExclusive;

    public SummingTask(List<Integer> numbers) {
        this(numbers, 0, numbers.size());
    }

    // This constructor is only used internally as part of the dividing process
    private SummingTask(List<Integer> numbers, int minInclusive, int maxExclusive) {
        this.numbers = numbers;
        this.minInclusive = minInclusive;
        this.maxExclusive = maxExclusive;
    }

    @Override
    public Integer compute() {
        if (maxExclusive - minInclusive > MAX_BATCH_SIZE) {
            // This is too big for a single batch, so we shall divide into two tasks
            int mid = (minInclusive + maxExclusive) / 2;
            SummingTask leftTask = new SummingTask(numbers, minInclusive, mid);
            SummingTask rightTask = new SummingTask(numbers, mid, maxExclusive);

            // Submit the left hand task as a new task to the same ForkJoinPool
            leftTask.fork();

            // Run the right hand task on the same thread and get the result
        }
    }
}
```

```

        int rightResult = rightTask.compute();

        // Wait for the left hand task to complete and get its result
        int leftResult = leftTask.join();

        // And combine the result
        return leftResult + rightResult;
    } else {
        // This is fine for a single batch, so we will run it here and now
        int sum = 0;
        for (int i = minInclusive; i < maxExclusive; i++) {
            sum += numbers.get(i);
        }
        return sum;
    }
}
}

```

このタスクのインスタンスを[ForkJoinPool](#)のインスタンスにすることができます。

```

// Because I am not specifying the number of threads
// it will create a thread for each available processor
ForkJoinPool pool = new ForkJoinPool();

// Submit the task to the pool, and get what is effectively the Future
ForkJoinTask<Integer> task = pool.submit(new SummingTask(numbers));

// Wait for the result
int result = task.join();

```

オンラインでフォーク/フレームワークによるプログラミングをむ

<https://riptutorial.com/ja/java/topic/4245/フォーク-フレームワークによるプログラミング>

117: プリミティブデータ

き

`byte`、`short`、`int`、`long`、`char`、`boolean`、`float`、`double`の8のプリミティブデータは、Javaプログラムでものデータをするです。

- `int aInt = 8;` //この`int`のはリテラルとばれます。
- `int hexInt = 0x1a;` // = 26; リテラルは、が`0x`の 16でできます。
- `int binInt = 0b11010;` // = 26; バイナリリテラルをすることもできます。は`0b`です。
- `long goodLong = 100000000000L;` //デフォルトでは、リテラルは`int`です。リテラルのに`L`をすることによって、リテラルがいことをコンパイラにえます。これがなければ、コンパイラは "Integer number too large" エラーをスローします。
- `double aDouble = 3.14;` //リテラルはデフォルトで`double`です。
- `float aFloat = 3.14F;` //デフォルトでは、このリテラルは`double` "のない" エラーがしていましたが、`F`をすることによってコンパイラに`float`をします。

Javaには、`boolean`、`byte`、`short`、`char`、`int`、`long`、`float`、`double` 8つのプリミティブデータがあります。のすべてのがであるこれは、すべてのをみ、そしてビルトインJavaでなをとっているオブジェクトタイプ/クラス; えば `String`、`Class` と `Throwable` およびそのサブクラスを。

プリミティブのすべての、、、、のは、なくともある`int`って、`short` への`shortint`すると、`byte` の`byte`、または`char` する`char`。そのをじのにしたいは、それをキャストしなければなりません。え

```
byte a = 1;
byte b = 2;
byte c = (byte) (a + b);
```

をキャストしないと、コンパイルエラーがします。

これは、[Java§2.11.1](#)ののにします。

コンパイラは、コンパイルまたはにこれらのを`int`のにするJavaマシンをして、`byte` および`short` のリテラルのロードをエンコードします。`boolean` と `char` のリテラルのロードは、コンパイルまたはにリテラルを`int`のにゼロするをしてエンコードされます。

[.]。したがって、の`boolean`、`byte`、`char`、`short` にするほとんどののは、`int` をすることによってしくされます。

このにあるは、そのセクションでもされています。

Java Virtual Machineの1バイトのオペコードサイズがえられているので、タイプをオペコードにエンコードすると、セットのにかかります。きがすべてのJavaマシンのデータをサポートしていれば、1 byteでできるのが1 byte。 [...]にして、サポートされていないデータとサポートされているデータをするために、々のをできます。

Examples

intプリミティブ

intのようなプリミティブデータは、それをしているにをしますが、Integerをしてされたはそのへのをします。

Java APIによると "Integerクラスは、オブジェクトのプリミティブintのをラップします。Integerのオブジェクトには、がintののフィールドがまれています。

デフォルトでは、intは32ビットきです。 -2^{31} のと $2^{31}-1$ のをできます。

```
int example = -42;
int myInt = 284;
int anotherInt = 73;

int addedInts = myInt + anotherInt; // 284 + 73 = 357
int subtractedInts = myInt - anotherInt; // 284 - 73 = 211
```

こののをするがあるは、わりにlongをするがあります。intののをえと、のオーバーフローがし、をえるがののサイトにされますのはのになります。は $((\text{value} - \text{MIN\_VALUE}) \% \text{RANGE}) + \text{MIN\_VALUE}$ 、 $((\text{value} + 2147483648) \% 4294967296) - 2147483648$

```
int demo = 2147483647; //maximum positive integer
System.out.println(demo); //prints 2147483647
demo = demo + 1; //leads to an integer overflow
System.out.println(demo); // prints -2147483648
```

intのとはのにあります。

```
int high = Integer.MAX_VALUE; // high == 2147483647
int low = Integer.MIN_VALUE; // low == -2147483648
```

intのデフォルトは0です。

```
int defaultInt; // defaultInt == 0
```

いプリミティブ

shortは16ビットのきです。 -2^{15} -32,768のと、 2^{15} -132,767のをち、

```
short example = -48;
```

```
short myShort = 987;
short anotherShort = 17;

short addedShorts = (short) (myShort + anotherShort); // 1,004
short subtractedShorts = (short) (myShort - anotherShort); // 970
```

shortのとはのにあります。

```
short high = Short.MAX_VALUE; // high == 32767
short low = Short.MIN_VALUE; // low == -32768
```

shortのデフォルトは0です。

```
short defaultShort; // defaultShort == 0
```

いプリミティブ

デフォルトでは、longは64ビットのきですJava 8では、きまたはなしのいずれでもかまいません。きでは、 -2^{63} および $2^{63}-1$ をでき、なしでは0および $2^{64}-1$ をできます

```
long example = -42;
long myLong = 284;
long anotherLong = 73;

//an "L" must be appended to the end of the number, because by default,
//numbers are assumed to be the int type. Appending an "L" makes it a long
//as 549755813888 ( $2^{39}$ ) is larger than the maximum value of an int ( $2^{31}-1$ ),
//"L" must be appended
long bigNumber = 549755813888L;

long addedLongs = myLong + anotherLong; // 284 + 73 = 357
long subtractedLongs = myLong - anotherLong; // 284 - 73 = 211
```

longのとはのにあります。

```
long high = Long.MAX_VALUE; // high == 9223372036854775807L
long low = Long.MIN_VALUE; // low == -9223372036854775808L
```

デフォルトのlong 0Lです

```
long defaultLong; // defaultLong == 0L
```

longリテラルのにされた"L"はをしませんが、としやすいので、をすることをおめします。

```
2L == 21; // true
```

JavaではIntegerオブジェクトのインスタンスが-128127のでキャッシュされます。ここではについてします https://blogs.oracle.com/darcy/entry/boxing_and_caches_integer_valueof

のがわかりました

```

Long val1 = 127L;
Long val2 = 127L;

System.out.println(val1 == val2); // true

Long val3 = 128L;
Long val4 = 128L;

System.out.println(val3 == val4); // false

```

2つのObject Longをにするには、のコードをしますJava 1.7。

```

Long val3 = 128L;
Long val4 = 128L;

System.out.println(Objects.equal(val3, val4)); // true

```

プリミティブlongをObject longとすると、2つのオブジェクトを==ですのようなのネガティブはしません。

booleanプリミティブ

booleanは、trueまたはfalse 2つのいずれかをできfalse

```

boolean foo = true;
System.out.println("foo = " + foo); // foo = true

boolean bar = false;
System.out.println("bar = " + bar); // bar = false

boolean notFoo = !foo;
System.out.println("notFoo = " + notFoo); // notFoo = false

boolean fooAndBar = foo && bar;
System.out.println("fooAndBar = " + fooAndBar); // fooAndBar = false

boolean fooOrBar = foo || bar;
System.out.println("fooOrBar = " + fooOrBar); // fooOrBar = true

boolean fooXorBar = foo ^ bar;
System.out.println("fooXorBar = " + fooXorBar); // fooXorBar = true

```

booleanのデフォルトはfalseです。

```

boolean defaultBoolean; // defaultBoolean == false

```

バイトプリミティブ

1 byteは8ビットのきです。 1127 -これは-2⁷⁻¹²⁸、び2⁷のをすることができます

```

byte example = -36;
byte myByte = 96;
byte anotherByte = 7;

```

```
byte addedBytes = (byte) (myByte + anotherByte); // 103
byte subtractedBytes = (byte) (myBytes - anotherByte); // 89
```

byteのとはのにありbyte。

```
byte high = Byte.MAX_VALUE; // high == 127
byte low = Byte.MIN_VALUE; // low == -128
```

byteのデフォルトは0です。

```
byte defaultByte; // defaultByte == 0
```

floatプリミティブ

float32ビットIEEE 754です。デフォルトでは、は2としてされます。 floatをするfloat、に10リテラルにfをします。

```
double doubleExample = 0.5; // without 'f' after digits = double
float floatExample = 0.5f; // with 'f' after digits = float

float myFloat = 92.7f; // this is a float...
float positiveFloat = 89.3f; // it can be positive,
float negativeFloat = -89.3f; // or negative
float integerFloat = 43.0f; // it can be a whole number (not an int)
float underZeroFloat = 0.0549f; // it can be a fractional value less than 0
```

は、5つのな、、、モジュラスをします。

は、エラーのとしてなるがあります。のは、わかりやすくみやすいようにめられていますつまり、のはには34.600002でした。

```
// addition
float result = 37.2f + -2.6f; // result: 34.6

// subtraction
float result = 45.1f - 10.3f; // result: 34.8

// multiplication
float result = 26.3f * 1.7f; // result: 44.71

// division
float result = 37.1f / 4.8f; // result: 7.729166

// modulus
float result = 37.1f % 4.8f; // result: 3.4999971
```

がバイナリでされるのため、くのはなをっていません。

```
float notExact = 3.1415926f;
System.out.println(notExact); // 3.1415925
```

ほとんどのアプリケーションで `float` を `float` はありませんが、10のななど、またはよりいがされるをするには、 `float` も `double` もしないでください。わりに、 `BigDecimal` クラスをするがあります。

`float` のデフォルトは `0.0f` です。

```
float defaultFloat;    // defaultFloat == 0.0f
```

`float` は、には1000の1のです。

`Float.POSITIVE_INFINITY`、 `Float.NEGATIVE_INFINITY`、 `Float.NaN` は `float` です。 `NaN` は、2つのなど、できないのをします。さらに、 `0f` と `-0f` はなるが、 `==` がする。

```
float f1 = 0f;
float f2 = -0f;
System.out.println(f1 == f2); // true
System.out.println(1f / f1); // Infinity
System.out.println(1f / f2); // -Infinity
System.out.println(Float.POSITIVE_INFINITY / Float.POSITIVE_INFINITY); // NaN
```

ダブルプリミティブ

`double` は、の64ビットIEEE 754です。

```
double example = -7162.37;
double myDouble = 974.21;
double anotherDouble = 658.7;

double addedDoubles = myDouble + anotherDouble; // 315.51
double subtractedDoubles = myDouble - anotherDouble; // 1632.91

double scientificNotationDouble = 1.2e-3;    // 0.0012
```

がされるのため、くのはなをっていません。

```
double notExact = 1.32 - 0.42; // result should be 0.9
System.out.println(notExact); // 0.90000000000000001
```

している `double` ほとんどのアプリケーションです、どちらも `float` も `double` など、なをするためにするがあります。わりに、 `BigDecimal` クラスをするがあります

`double` のデフォルトは `0.0d` です。

```
public double defaultDouble;    // defaultDouble == 0.0
```

`Double.POSITIVE_INFINITY`、 `Double.NEGATIVE_INFINITY`、 `Double.NaN` は `double` です。 `NaN` は、2つのなど、できないのをします。さらに、 `0d` と `-0d` はなりませんが、 `==` が `-0d` ます。

```
double d1 = 0d;
```

```
double d2 = -0d;
System.out.println(d1 == d2); // true
System.out.println(1d / d1); // Infinity
System.out.println(1d / d2); // -Infinity
System.out.println(Double.POSITIVE_INFINITY / Double.POSITIVE_INFINITY); // NaN
```

char プリミティブ

char はの16ビットUnicodeをできます。リテラルはでまれています

```
char myChar = 'u';
char myChar2 = '5';
char myChar3 = 65; // myChar3 == 'A'
```

\u0000 10では0、ヌルともばれます と \uffff 65,535のを \uffff ます。

char のデフォルトは \u0000 です。

```
char defaultChar; // defaultChar == \u0000
```

・ value をするためには、エスケープシーケンスバックスラッシュでまるをするがあります

```
char singleQuote = '\'';
```

そののエスケープシーケンスもあります

```
char tab = '\t';
char backspace = '\b';
char newline = '\n';
char carriageReturn = '\r';
char formfeed = '\f';
char singleQuote = '\'';
char doubleQuote = '\"'; // escaping redundant here; '"' would be the same; however still
allowed
char backslash = '\\';
char unicodeChar = '\uXXXX' // XXXX represents the Unicode-value of the character you want to
display
```

のUnicodeの char をできます。

```
char heart = '\u2764';
System.out.println(Character.toString(heart)); // Prints a line containing "♥".
```

char にすることもできます。たとえば、すべてのをりしするには、のようにします。

```
for (int i = 0; i <= 26; i++) {
    char letter = (char) ('a' + i);
    System.out.println(letter);
}
```

の

Javaおよびのほとんどののは、 2 のとばれるにのをします。

n ビットをするデータののバイナリでは、はのようにエンコードされます。

の $n-1$ ビットは、での x する。もなは、 s ビットをします。これらのビットがすは

$$x - s * 2^{n-1}$$

すなわち、ビットが1であれば、のビット $2^{n-2} + 2^{n-3} + \dots + 2^1 + 2^0 = 2^{n-1} - 1$ することができ
るよりも1だけきい $2^{n-2} + 2^{n-3} + \dots + 2^1 + 2^0 = 2^{n-1} - 1$ からのののバイナリされる- $2^{n-1} S$
 $= 1$ 、 $X = 0$ の $n-1$ $2 - 1 S = 0$ 、 $X = 2^{N-1} - 1$ 。

これにはいがあり、バイナリをの 2 のようにすることができます

$$\begin{aligned} v1 &= x1 - s1 * 2^{n-1} \\ v2 &= x2 - s2 * 2^{n-1} \end{aligned}$$

s1	s2	x1 + x2 オーバーフロー	
0	0	いいえ	$x1 + x2 = v1 + v2$
0	0	はい	データでするにはきすぎますオーバーフロー
0	1	いいえ	$x1 + x2 - 2^{n-1} = x1 + x2 - s2 * 2^{n-1} = v1 + v2$
0	1	はい	$(x1 + x2) \bmod 2^{n-1} = x1 + x2 - 2^{n-1} = v1 + v2$
1	0	*	をサマリを
1	1	いいえ	データ $x1 + x2 - 2^n - 2^{n-1}$; アンダーフローでするにはさすぎます。
1	1	はい	$\begin{aligned} (x1 + x2) \bmod 2^{n-1} - 2^{n-1} &= (x1 + x2 - 2^{n-1}) - 2^{n-1} \\ &= (x1 - s1 * 2^{n-1}) + (x2 - s2 * 2^{n-1}) \\ &= v1 + v2 \end{aligned}$

このは、すなわち、のの 2 をにつけることをにすることにされたい

にビットをえるとすべてのビットが1になることにしてください。に、1をしてのオーバーフローをい、ニュートラル0すべてのビット0をします。

したがって、`i`ののは、`int`へのなをして

```
(~i) + 1
```

の0 byte をる

0をしたは11111111です。1をえると100000000 9ビットのがられます。byteは8ビットしかできないため、のはりてられ、は00000000

の	プロセス	
000000000		-011111111
11111111	バイナリに1をえる	100000000
100000000	8ビットにりてる	00000000-0は0にしい

プリミティブとボックスプリミティブのメモリ

プリミティブ	ボックスタイプ	プリミティブ/ボックスのメモリサイズ
ブール	ブール	1バイト/ 16バイト
バイト	バイト	1バイト/ 16バイト
ショート	ショート	2バイト/ 16バイト
チャー	チャー	2バイト/ 16バイト
int		4バイト/ 16バイト
いです	いです	8バイト/ 16バイト
く	く	4バイト/ 16バイト
ダブル	ダブル	8バイト/ 16バイト

ボックスされたオブジェクトはにとメモリのに8バイトをとし、オブジェクトのサイズはに8のであるため、ボックスされたはすべて16バイトをとします。また、ボックスオブジェクトのはJVMとJVMオプションにじて、の4つのまたは8バイトをめるをすることをいします。

データでは、メモリがパフォーマンスにきなをえるがあります。をすると、メモリはさらにえま

す `float[5]` には32バイトしかありません。5つのヌルをする `Float[5]` 112バイトをとしますポインタなしの64ビットでは、これは152バイトにします。

ボックスされたのキャッシュ

ボックスされたのスペースオーバーヘッドは、ボックスされたのキャッシュによってあるできます。ボックスされたのには、インスタンスのキャッシュをするものがあります。たとえば、デフォルトでは、`Integer` クラスはインスタンスをキャッシュして、 $-128 + 127$ のをします。ただし、メモリのなによってするコストはされません。

autoboxing または `valueOf(primitive)` メソッドをびすことによって `boxed` のインスタンスをする、ランタイムシステムはキャッシュされたのをみます。アプリケーションがキャッシュされたのくのを、ボックスされたをすることによるメモリのペナルティがにされます。かに、"で" ボックスされたバリュー・インスタンスをするは、`new` ではなく `valueOf` をするがよいでしょう。ただし、のがキャッシュされたにないは、`new` をびしてキャッシュルックアップをするがなくなるがあります `new` ではにしいインスタンスがされます。

プリミティブの

Javaでは、とののがです。また、すべてのがUnicodeエンコーディングのにするため、`char` はとにできます。`boolean` は、のプリミティブデータとのでできないのプリミティブデータです。

コンバージョンには、コンバージョンのとコンバージョンの の2があります。

は、1つのデータのが、のデータよりもくのビットをめるのデータのにされるです。この、データのはありません。

これにして、1つのデータのが、のデータよりもないビットをめるのデータのにされるとき、がくなります。この、データのがするがあります。

Javaはにをします。しかし、をくしたいはデータがわれないことがかな、`cast` ばれるをしてJavaにをさせることができます。

```
int a = 1;
double d = a;    // valid conversion to double, no cast needed (widening)
```

ナロー

```
double d = 18.96
int b = d;        // invalid conversion to int, will throw a compile-time error
int b = (int) d;  // valid conversion to int, but result is truncated (gets rounded down)
                // This is type-casting
                // Now, b = 18
```

プリミティブチートシート

すべてのプリミティブのサイズとのをす。

データ・ タイプ		の	デフ ォ ルト
ブール	なし	と	
バイト	8ビット とき	$1 - 2^{-27}$	0
-128+127			
シヨー ト	16ビット とき	$1 - 2^{-15}$	0
-32,768+ 32,767			
int	32ビット とき	$1 - 2^{-31}$	0
-2,147,483,648+ 2,147,483,647			
いです	64ビット ト	$1 - 2^{-63}$	0L
-9,223,372,036,854,775,808~9,223,372,036,854,775,807			
く	32ビット ト	1.401298464e-453.402823466e + 38または	0.0F
ダブル	64ビット ト	4.94065645841246544e-324dから1.79769313486231570e + 308dまたは	0.0D
チャー	16ビット トなし	0から $2^{16} - 1$	0
065535			

ノート

1. Javaでは、き `byte` から `long` は22をし、はIEE 754 2をすることがづけられています。
2. Java 8では、`int` および `long` なしをするメソッドをしています。これらのメソッドでは、プログラムはタイプのをなしとてうことができますが、タイプはきタイプのままです。
3. にしたのはです。つまり、のよりもがります。のは1.175494351e-38と2.2250738585072014e-308です
4. `char` はUnicode / UTF-16 コードをします。
5. `boolean` は1ビットのしかまれていませんが、メモリのサイズはJavaマシンのによってなりま

す [bool](#) を。

オンラインでプリミティブデータをむ <https://riptutorial.com/ja/java/topic/148/プリミティブデータ>

118: プログラムによるの

`BufferedImage.getGraphics()` に `Graphics2D` をします。

`VolatileImage` をすると、のがにすることがありますが、そのもあります。そのはいつでもわれ、ゼロからするがあります。

Examples

なイメージをプログラムでしてする

```
class ImageCreationExample {

    static Image createSampleImage() {
        // instantiate a new BufferedImage (subclass of Image) instance
        BufferedImage img = new BufferedImage(640, 480, BufferedImage.TYPE_INT_ARGB);

        //draw something on the image
        paintOnImage(img);

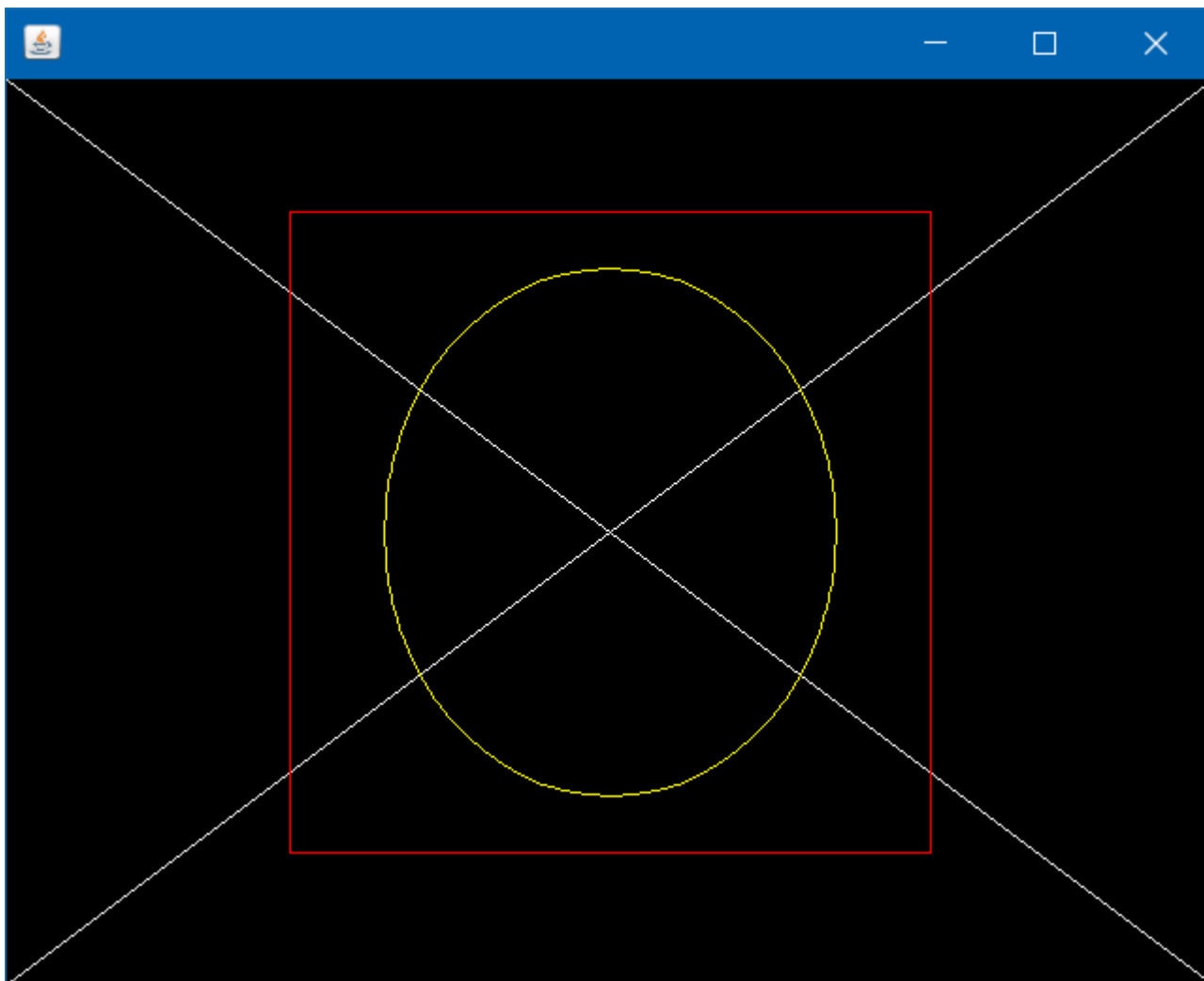
        return img;
    }

    static void paintOnImage(BufferedImage img) {
        // get a drawable Graphics2D (subclass of Graphics) object
        Graphics2D g2d = (Graphics2D) img.getGraphics();

        // some sample drawing
        g2d.setColor(Color.BLACK);
        g2d.fillRect(0, 0, 640, 480);
        g2d.setColor(Color.WHITE);
        g2d.drawLine(0, 0, 640, 480);
        g2d.drawLine(0, 480, 640, 0);
        g2d.setColor(Color.YELLOW);
        g2d.drawOval(200, 100, 240, 280);
        g2d.setColor(Color.RED);
        g2d.drawRect(150, 70, 340, 340);

        // drawing on images can be very memory-consuming
        // so it's better to free resources early
        // it's not necessary, though
        g2d.dispose();
    }

    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Image img = createSampleImage();
        ImageIcon icon = new ImageIcon(img);
        frame.add(new JLabel(icon));
        frame.pack();
        frame.setVisible(true);
    }
}
```



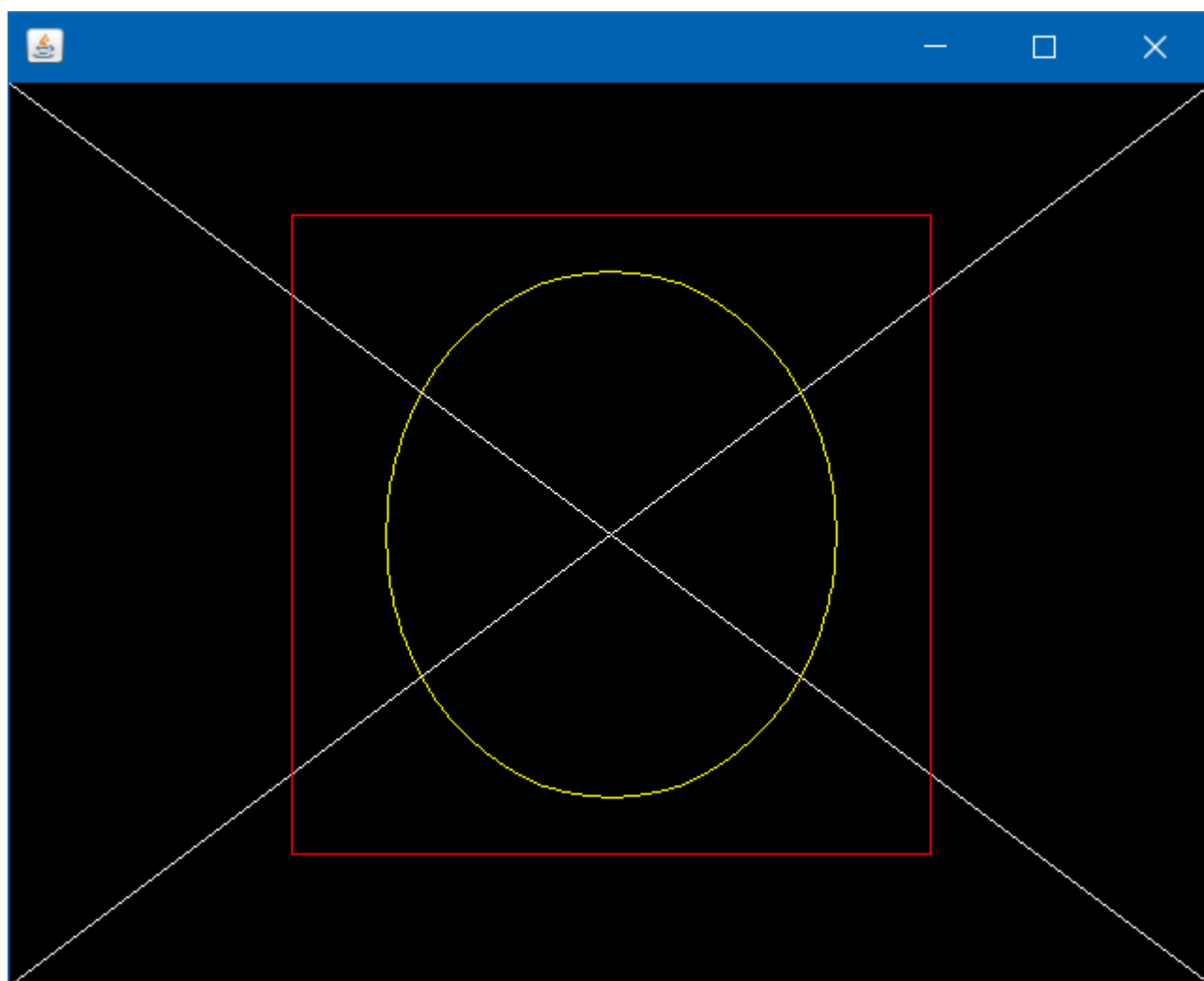
イメージをディスクにする

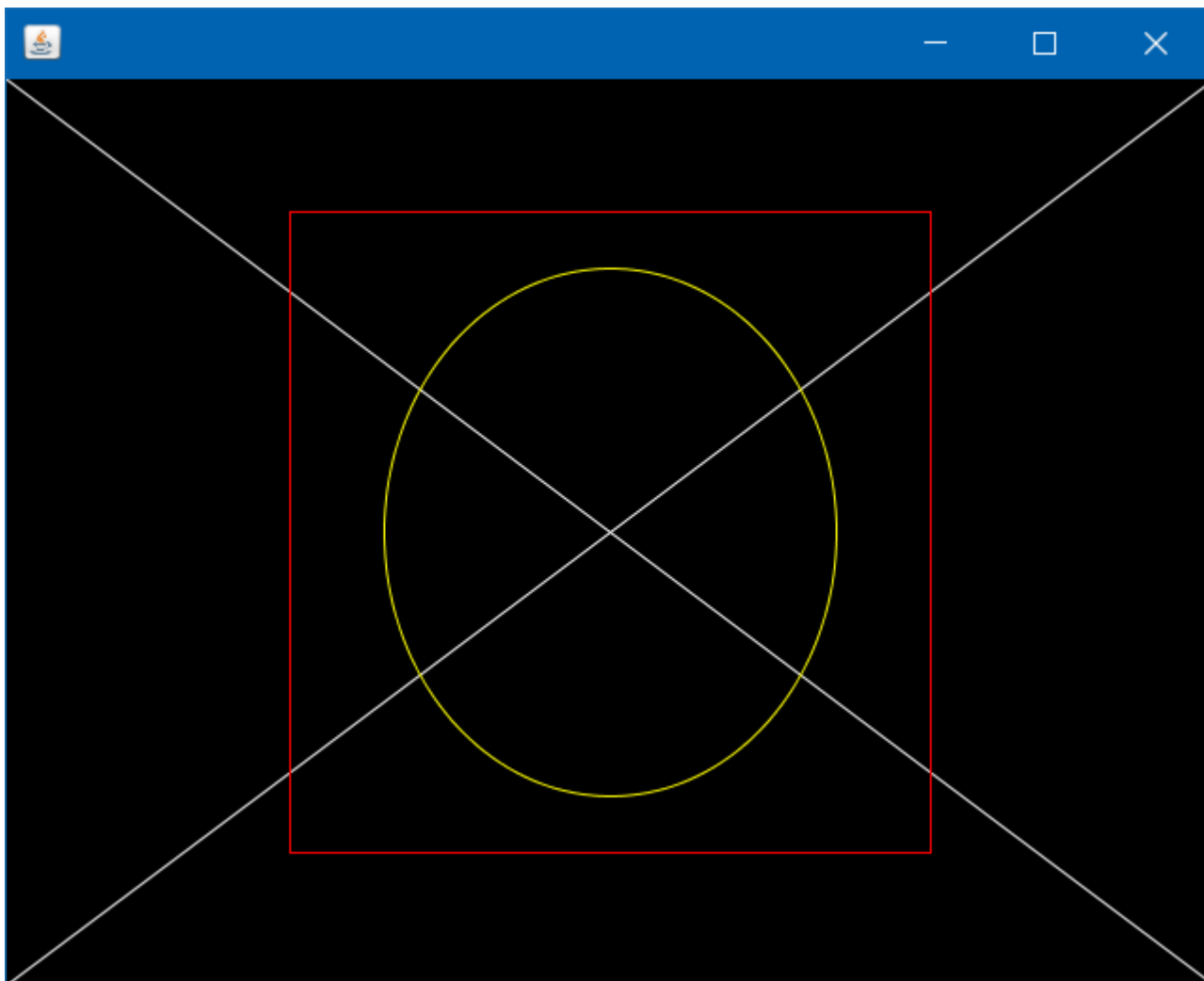
```
public static void saveImage(String destination) throws IOException {  
    // method implemented in "Creating a simple image Programmatically and displaying it"  
    example  
    BufferedImage img = createSampleImage();  
  
    // ImageIO provides several write methods with different outputs  
    ImageIO.write(img, "png", new File(destination));  
}
```

のレンダリングをする

```
static void setupQualityHigh(Graphics2D g2d) {  
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);  
    g2d.setRenderingHint(RenderingHints.KEY_RENDERING, RenderingHints.VALUE_RENDER_QUALITY);  
    // many other RenderingHints KEY/VALUE pairs to specify  
}  
  
static void setupQualityLow(Graphics2D g2d) {  
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_OFF);  
    g2d.setRenderingHint(RenderingHints.KEY_RENDERING, RenderingHints.VALUE_RENDER_SPEED);  
}
```

サンプルイメージのとスピードのレンダリングの





BufferedImage クラスをしたイメージの

```
int width = 256; //in pixels
int height = 256; //in pixels
BufferedImage image = new BufferedImage(width, height, BufferedImage.TYPE_4BYTE_ABGR);
//BufferedImage.TYPE_4BYTE_ABGR - store RGB color and visibility (alpha), see javadoc for more
info

Graphics g = image.createGraphics();

//draw whatever you like, like you would in a drawComponent(Graphics g) method in an UI
application
g.setColor(Color.RED);
g.fillRect(20, 30, 50, 50);

g.setColor(Color.BLUE);
g.drawOval(120, 120, 80, 40);

g.dispose(); //dispose graphics objects when they are no longer needed

//now image has programmatically generated content, you can use it in graphics.drawImage() to
draw it somewhere else
//or just simply save it to a file
ImageIO.write(image, "png", new File("myimage.png"));
```



BufferedImageでののと

```
BufferedImage cat = ImageIO.read(new File("cat.jpg")); //read existing file

//modify it
Graphics g = cat.createGraphics();
g.setColor(Color.RED);
g.drawString("Cat", 10, 10);
g.dispose();

//now create a new image
BufferedImage cats = new BufferedImage(256, 256, BufferedImage.TYPE_4BYTE_ABGR);

//and draw the old one on it, 16 times
g = cats.createGraphics();
for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 4; j++) {
        g.drawImage(cat, i * 64, j * 64, null);
    }
}

g.setColor(Color.BLUE);
g.drawRect(0, 0, 255, 255); //add some nice border
g.dispose(); //and done

ImageIO.write(cats, "png", new File("cats.png"));
```

のcatファイル



されたファイル



BufferedImageで々のピクセルのをする

```
BufferedImage image = new BufferedImage(256, 256, BufferedImage.TYPE_INT_ARGB);

//you don't have to use the Graphics object, you can read and set pixel color individually
for (int i = 0; i < 256; i++) {
    for (int j = 0; j < 256; j++) {
        int alpha = 255; //don't forget this, or use BufferedImage.TYPE_INT_RGB instead
        int red = i; //or any formula you like
        int green = j; //or any formula you like
        int blue = 50; //or any formula you like
        int color = (alpha << 24) | (red << 16) | (green << 8) | blue;
        image.setRGB(i, j, color);
    }
}

ImageIO.write(image, "png", new File("computed.png"));
```



BufferedImageをどのようにスケールするか

```
/**
 * Resizes an image using a Graphics2D object backed by a BufferedImage.
 * @param srcImg - source image to scale
 * @param w - desired width
```

```

* @param h - desired height
* @return - the new resized image
*/
private BufferedImage getScaledImage(Image srcImg, int w, int h){

    //Create a new image with good size that contains or might contain arbitrary alpha values
    between and including 0.0 and 1.0.
    BufferedImage resizedImg = new BufferedImage(w, h, BufferedImage.TRANSLUCENT);

    //Create a device-independant object to draw the resized image
    Graphics2D g2 = resizedImg.createGraphics();

    //This could be changed, Cf. http://stackoverflow.com/documentation/java/5482/creating-
    images-programmatically/19498/specifying-image-rendering-quality
    g2.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
    RenderingHints.VALUE_INTERPOLATION_BILINEAR);

    //Finally draw the source image in the Graphics2D with the desired size.
    g2.drawImage(srcImg, 0, 0, w, h, null);

    //Disposes of this graphics context and releases any system resources that it is using
    g2.dispose();

    //Return the image used to create the Graphics2D
    return resizedImg;
}

```

オンラインでプログラムによるのをむ <https://riptutorial.com/ja/java/topic/5482/プログラムによるの>

119: プロセス

APIでは、バージョン1.5、 `ProcessBuilder.start()` をしてプロセスをすることをおめします。

もうつの `waitFor` は、 `waitFor` によってされたがされているプログラム/スクリプトにしていることです。たとえば、 **calc.exe** によってされるコードは **notepad.exe** とはなりません。

Examples

なJavaバージョン<1.5

このでは、ウィンドウのをびします。コードは、びされるプログラム/スクリプトにじてすることにすることがです。

```
package process.example;

import java.io.IOException;

public class App {

    public static void main(String[] args) {
        try {
            // Executes windows calculator
            Process p = Runtime.getRuntime().exec("calc.exe");

            // Wait for process until it terminates
            int exitCode = p.waitFor();

            System.out.println(exitCode);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

ProcessBuilder クラスの

ProcessBuilderクラスをすると、コマンドラインからにコマンドをできます。なのは、するコマンドをすることです。コマンドをするには、にProcessBuilderインスタンスでstartメソッドをびします。

Add.exeという2つのをとり、それらをするプログラムがある、コードはのようになります。

```
List<String> cmds = new ArrayList<>();
cmds.add("Add.exe"); //the name of the application to be run
cmds.add("1"); //the first argument
cmds.add("5"); //the second argument
```

```

ProcessBuilder pb = new ProcessBuilder(cmds);

//Set the working directory of the ProcessBuilder so it can find the .exe
//Alternatively you can just pass in the absolute file path of the .exe
File myWorkingDirectory = new File(yourFilePathNameGoesHere);
pb.workingDirectory(myWorkingDirectory);

try {
    Process p = pb.start();
} catch (IOException e) {
    e.printStackTrace();
}

```

にめておくべきこと

- コマンドのはすべてでなければなりません
- コマンドラインでプログラムをびしたつまり、.exeのがののにかれない、コマンドはのでなければなりません
- ディレクトリをするときは、ファイルだけでなくFileオブジェクトをすがあります

ブロッキングとブロッキングのコール

に、コマンドラインをびすとき、プログラムはコマンドをしてからをします。

しかし、びされたプログラムがするのをってから、のをけることができますex。びされたプログラムはデータをファイルにきし、プログラムはそのデータにアクセスするがあります。

これは、されたProcessインスタンスからwaitFor() メソッドをびすことでにできます。

```

//code setting up the commands omitted for brevity...

ProcessBuilder pb = new ProcessBuilder(cmds);

try {
    Process p = pb.start();
    p.waitFor();
} catch (IOException e) {
    e.printStackTrace();
} catch (InterruptedException e) {
    e.printStackTrace();
}

//more lines of code here...

```

ch.vorburger.exec

raw java.lang.ProcessBuilder APIをしてJavaからプロセスをするのはしです。 [Apache Commons Exec ライブラリ](#)をうとしになります。 [ch.vorburger.exec ライブラリ](#)は、さらにCommons Execをして、にになりました。

```

ManagedProcess proc = new ManagedProcessBuilder("path-to-your-executable-binary")
    .addArgument("arg1")

```

```

        .addArgument("arg2")
        .setWorkingDirectory(new File("/tmp"))
        .setDestroyOnShutdown(true)
        .setConsoleBufferMaxLines(7000)
        .build();

proc.start();
int status = proc.waitForExit();
int status = proc.waitForExitMaxMsOrDestroy(3000);
String output = proc.getConsole();

proc.startAndWaitForConsoleMessageMaxMs("started!", 7000);
// use service offered by external process...
proc.destroy();

```

とし**Runtime.exec**、**Process**、および**ProcessBuilder**はシェルをしていません

`Runtime.exec(String ...)`と`Runtime.exec(String)`メソッドをすると、プロセス¹のようにコマンドをすることができます。のバージョンでは、コマンドとコマンドをの々のとしてし、JavaランタイムはOSランタイムシステムにコマンドをするようにします。2のバージョンはいがですが、いくつかのとしがあります。

まず、にされている`exec(String)`をにします。

```

Process p = Runtime.exec("mkdir /tmp/testDir");
p.waitFor();
if (p.exitValue() == 0) {
    System.out.println("created the directory");
}

```

パスのスペース

のをして、のディレクトリをできるとします。

```

Process p = Runtime.exec("mkdir " + dirPath);
// ...

```

これはしますが、`dirPath`が`"/home/user/My Documents"`などのはします。は、`exec(String)`がをすだけでをコマンドとにすることです。コマンド

```
"mkdir /home/user/My Documents"
```

されます

```
"mkdir", "/home/user/My", "Documents"
```

これは2つではなく1つのをとするため、`"mkdir"`コマンドがするとなります。

これにして、のプログラマはパスののにをしようとします。これはどちらもしません

```
"mkdir \"/home/user/My Documents\""
```

されます

```
"mkdir", "\"/home/user/My", "Documents\""
```

スペースを "" しようとしてされたのは、ののことにわれます。かに、たちがスペースをしたりエスケープしたりするものはすべてするでしょう。

こののにするは、`exec(String ...)` オーバーロードをすることです。

```
Process p = Runtime.exec("mkdir", dirPath);  
// ...
```

これは、`dirPath` にがまれているとします。これは、`exec` このオーバーロードがをしようとしなためです。はそのままOS `exec` システムコールにされます。

リダイレクト、パイプライン、そののシェル

コマンドのまたはをリダイレクトする、またはパイプラインをするとします。えば

```
Process p = Runtime.exec("find / -name *.java -print 2>/dev/null");
```

または

```
Process p = Runtime.exec("find source -name *.java | xargs grep package");
```

のは、ファイルシステムのすべてのJavaファイルのをリストし、2のは、`package`²を「ソース」ツリーのJavaファイルにし`package`。

これらはりにしません。のケースでは、`"find"` コマンドは `"2 / dev / null"` をコマンドとしてされます。これはリダイレクトとしてされません。2のでは、パイプ `"|"` とそれにくが `"find"` コマンドにえられます。

ここでののは、`exec` メソッドと `ProcessBuilder` がシェルのをしていないことです。これには、リダイレクション、パイプライン、グロピングなどがまれます。

いくつかのケースなりダイレクションなどでは、`ProcessBuilder` をしてにのをできます。しかし、これはではありません。のアプローチは、シェルでコマンドラインをすることです。えば

```
Process p = Runtime.exec("bash", "-c",  
                          "find / -name *.java -print 2>/dev/null");
```

または

```
Process p = Runtime.exec("bash", "-c",
```

```
"find source -name \\.java | xargs grep package");
```

しかし、2のでは、ワイルドカードをシェルではなく "find"によってさせたいので、ワイルドカード "*"をエスケープするがあることにしてください。

シェルみみコマンドがしない

のは、UNIXのようなシェルをつシステムではしないものとします。

```
Process p = Runtime.exec("cd", "/tmp"); // Change java app's home directory
```

または

```
Process p = Runtime.exec("export", "NAME=value"); // Export NAME to the java app's environment
```

これがうまくいかないはいくつかあります

1. "cd"と "export"コマンドはシェルのみみコマンドです。のファイルとしてしません。
2. シェルビルトインがうべきことえば、ディレクトリの、のをうためには、そのがするをするがあります。のアプリケーションJavaアプリケーションをむの、はアプリケーションプロセスにけられます。たとえば、"cd"コマンドをするプロセスは、"java"プロセスのディレクトリをできませんでした。に、1つの`exec`プロセスは、それにくプロセスのディレクトリをすることはできません。

このはすべてのシェルみみコマンドにされます。

1 - `ProcessBuilder`もできますが、これはこののポイントにはありません。

2 - これはちょっとラフでができていますが、このアプローチのはこのにはありません。

オンラインでプロセスをむ <https://riptutorial.com/ja/java/topic/4682/プロセス>

120: プロパティクラス

き

プロパティオブジェクトには、キーとのペアがともとしてされます。 `java.util.Properties` クラスは、`Hashtable` のサブクラスです。

これは、プロパティキーについてプロパティをするためにできます。 `Properties` クラスは、プロパティファイルからデータをし、データをプロパティファイルにするメソッドをします。また、システムのプロパティをするためにできます。

プロパティファイルの

がプロパティファイルからされた、コンパイルはありません。

- プロパティファイルではのようになります。
- キー=
- コメント

`Properties` オブジェクトは、キーとがにつて `String` である `Map` です。 `Map` のメソッドをしてデータにアクセスすることはできますが、はよりなメソッド `getProperty`、`setProperty`、および `stringPropertyNames` がわりにされます。

プロパティは、なテキストファイルである `Java` プロパティファイルににされます。それらののは、`Properties.load` メソッドでにされています。すれば

- キー/のペアは、のテキストのであるしい=、またはコロン ; キーとの。 `equals` またはコロンにはのがあり、されます。
- するはにされ、のはにまれます。
- バックスラッシュをしてのの `\u` をくをエスケープすることができます。
- のバックスラッシュは、のがののきであることをします。しかし、すべてのとに、ののはされます。
- `Java` ソースコードとに、 `\u` く 4 の 16 は `UTF-16` をします。

`java.util.ResourceBundle` のような `Java SE` のをむほとんどのフレームワークは、プロパティファイルを `InputStream` としてロードします。 `InputStream` からプロパティファイルをみむとき、そのファイルには `ISO 8859-10-255` のののみがまれます。ののをしなければならぬ `\u` エスケープ。ただし、のエンコーディングでテキストファイルをし、それをエスケープするために `native2ascii` ツールすべての `JDK` にをすることができます。

のコードをむプロパティファイルをロードする、する `Charset` について `Reader` `InputStreamReader` などをするり、どのエンコーディングでもかまいません。レガシーロード `InputStream` メソッドのわりに `loadReader` をしてファイルをロードできます。

プロパティをなXMLファイルにすることもできます。これにより、ファイルでエンコーディングをできます。このようなファイルは、[loadFromXML](#)メソッドでロードできます。このようなXMLファイルのをするDTDは、<http://java.sun.com/dtd/properties.dtd>にあります。

Examples

ロードプロパティ

アプリケーションにバンドルされているプロパティファイルをロードするには

```
public class Defaults {

    public static Properties loadDefaults() {
        try (InputStream bundledResource =
            Defaults.class.getResourceAsStream("defaults.properties")) {

            Properties defaults = new Properties();
            defaults.load(bundledResource);
            return defaults;
        } catch (IOException e) {
            // Since the resource is bundled with the application,
            // we should never get here.
            throw new UncheckedIOException(
                "defaults.properties not properly packaged"
                + " with application", e);
        }
    }
}
```

プロパティファイル **caveat** の

にであるこれらの2つのプロパティファイルをしくてみましょう

1 # Example 1	1 # Example 2
2	2
3 lastName=Smith	3 lastName=Smith
4	4

らはにではありません

1 # Example 1CRLE	1 # Example 2CRLE
2 CRLE	2 CRLE
3 lastName=SmithCRLE	3 lastName=Smith CRLE
4	4

スクリーンショットはNotepad ++からのものです

のがされるための`lastName` であろう"Smith"のケースとで"Smith "のです。

にまれに、これはユーザーがしているもので、これがなぜ`Properties`クラスのデフォルトのである

のかすることはできません。ただし、このをするバージョンの`Properties`をするのはです。のクラス、**TrimmedProperties**はそれだけを行います。これは、の`Properties`クラスのドロップインです。

```
import java.io.FileInputStream;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.Reader;
import java.util.Map.Entry;
import java.util.Properties;

/**
 * Properties class where values are trimmed for trailing whitespace if the
 * properties are loaded from a file.
 *
 * <p>
 * In the standard {@link java.util.Properties Properties} class trailing
 * whitespace is always preserved. When loading properties from a file such
 * trailing whitespace is almost always <i>unintentional</i>. This class fixes
 * this problem. The trimming of trailing whitespace only takes place if the
 * source of input is a file and only where the input is line oriented (meaning
 * that for example loading from XML file is <i>not</i> changed by this class).
 * For this reason this class is almost in all cases a safe drop-in replacement
 * for the standard <tt>Properties</tt>
 * class.
 *
 * <p>
 * Whitespace is defined here as any of space (U+0020) or tab (U+0009).
 *
 */
public class TrimmedProperties extends Properties {

    /**
     * Reads a property list (key and element pairs) from the input byte stream.
     *
     * <p>Behaves exactly as {@link java.util.Properties#load(java.io.InputStream) }
     * with the exception that trailing whitespace is trimmed from property values
     * if <tt>inStream</tt> is an instance of <tt>FileInputStream</tt>.
     *
     * @see java.util.Properties#load(java.io.InputStream)
     * @param inStream the input stream.
     * @throws IOException if an error occurred when reading from the input stream.
     */
    @Override
    public void load(InputStream inStream) throws IOException {
        if (inStream instanceof FileInputStream) {
            // First read into temporary props using the standard way
            Properties tempProps = new Properties();
            tempProps.load(inStream);
            // Now trim and put into target
            trimAndLoad(tempProps);
        } else {
            super.load(inStream);
        }
    }

    /**
     * Reads a property list (key and element pairs) from the input character stream in a
     * simple line-oriented format.
     */
}
```

```

*
* <p>Behaves exactly as {@link java.util.Properties#load(java.io.Reader)}
* with the exception that trailing whitespace is trimmed on property values
* if <tt>reader</tt> is an instance of <tt>FileReader</tt>.
*
* @see java.util.Properties#load(java.io.Reader) }
* @param reader the input character stream.
* @throws IOException if an error occurred when reading from the input stream.
*/
@Override
public void load(Reader reader) throws IOException {
    if (reader instanceof FileReader) {
        // First read into temporary props using the standard way
        Properties tempProps = new Properties();
        tempProps.load(reader);
        // Now trim and put into target
        trimAndLoad(tempProps);
    } else {
        super.load(reader);
    }
}

private void trimAndLoad(Properties p) {
    for (Entry<Object, Object> entry : p.entrySet()) {
        if (entry.getValue() instanceof String) {
            put(entry.getKey(), trimTrailing((String) entry.getValue()));
        } else {
            put(entry.getKey(), entry.getValue());
        }
    }
}

/**
 * Trims trailing space or tabs from a string.
 *
 * @param str
 * @return
 */
public static String trimTrailing(String str) {
    if (str != null) {
        // read str from tail until char is no longer whitespace
        for (int i = str.length() - 1; i >= 0; i--) {
            if ((str.charAt(i) != ' ') && (str.charAt(i) != '\t')) {
                return str.substring(0, i + 1);
            }
        }
    }
    return str;
}
}

```

XMLとしてのプロパティの

XMLファイルへのプロパティの

プロパティファイルをXMLファイルとしてするは、`.properties`ファイルとしてするとよくてい
ます。 `store()` をするわりに、 `storeToXML()` をします。

```

public void saveProperties(String location) throws IOException{
    // make new instance of properties
    Properties prop = new Properties();

    // set the property values
    prop.setProperty("name", "Steve");
    prop.setProperty("color", "green");
    prop.setProperty("age", "23");

    // check to see if the file already exists
    File file = new File(location);
    if (!file.exists()){
        file.createNewFile();
    }

    // save the properties
    prop.storeToXML(new FileOutputStream(file), "testing properties with xml");
}

```

ファイルをくと、のようになります。

```

1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
3  <properties>
4  <comment>testing properties with xml</comment>
5  <entry key="age">23</entry>
6  <entry key="color">green</entry>
7  <entry key="name">Steve</entry>
8  </properties>
9

```

XML ファイルからのプロパティのロード

このファイルを `properties` として `load()` するには、の `properties` ファイルでする `load()` わりに `loadFromXML()` をびすがあります。

```

public static void loadProperties(String location) throws FileNotFoundException, IOException{
    // make new properties instance to load the file into
    Properties prop = new Properties();

    // check to make sure the file exists
    File file = new File(location);
    if (file.exists()){
        // load the file
        prop.loadFromXML(new FileInputStream(file));

        // print out all the properties
        for (String name : prop.stringPropertyNames()){
            System.out.println(name + "=" + prop.getProperty(name));
        }
    } else {
        System.err.println("Error: No file found at: " + location);
    }
}

```

このコードをすると、コンソールにのコードがされます。

```
age=23  
color=green  
name=Steve
```

オンラインでプロパティクラスをむ <https://riptutorial.com/ja/java/topic/576/プロパティクラス>

121: ベンチマーク

き

Javaでのパフォーマンスベンチマークのは、とで`System.currentTimeMillis()`をし、そのをするほどではありません。なパフォーマンスベンチマークをくには、なツールをするがあります。

Examples

なJMHの

なベンチマークテストをくためのツールの1つはJMHです。 `HashSet` と `TreeSet` ののパフォーマンスをしたいとしましょう。

JMHをプロジェクトにりむもなは、MavenとShade Pluginをすることです。また、 JMHのから `pom.xml` をることができます。

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-shade-plugin</artifactId>
      <version>3.0.0</version>
      <executions>
        <execution>
          <phase>package</phase>
          <goals>
            <goal>shade</goal>
          </goals>
          <configuration>
            <finalName>/benchmarks</finalName>
            <transformers>
              <transformer
                implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                <mainClass>org.openjdk.jmh.Main</mainClass>
              </transformer>
            </transformers>
            <filters>
              <filter>
                <artifact>*:*</artifact>
                <excludes>
                  <exclude>META-INF/*.SF</exclude>
                  <exclude>META-INF/*.DSA</exclude>
                  <exclude>META-INF/*.RSA</exclude>
                </excludes>
              </filter>
            </filters>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

```

    </plugins>
</build>

<dependencies>
  <dependency>
    <groupId>org.openjdk.jmh</groupId>
    <artifactId>jmh-core</artifactId>
    <version>1.18</version>
  </dependency>
  <dependency>
    <groupId>org.openjdk.jmh</groupId>
    <artifactId>jmh-generator-annprocess</artifactId>
    <version>1.18</version>
  </dependency>
</dependencies>

```

この、ベンチマーククラスをくがあります

```

package benchmark;

import org.openjdk.jmh.annotations.*;
import org.openjdk.jmh.infra.Blackhole;

import java.util.HashSet;
import java.util.Random;
import java.util.Set;
import java.util.TreeSet;
import java.util.concurrent.TimeUnit;

@State(Scope.Thread)
public class CollectionFinderBenchmarkTest {
    private static final int SET_SIZE = 10000;

    private Set<String> hashSet;
    private Set<String> treeSet;

    private String stringToFind = "8888";

    @Setup
    public void setupCollections() {
        hashSet = new HashSet<>(SET_SIZE);
        treeSet = new TreeSet<>();

        for (int i = 0; i < SET_SIZE; i++) {
            final String value = String.valueOf(i);
            hashSet.add(value);
            treeSet.add(value);
        }

        stringToFind = String.valueOf(new Random().nextInt(SET_SIZE));
    }

    @Benchmark
    @BenchmarkMode(Mode.AverageTime)
    @OutputTimeUnit(TimeUnit.NANOSECONDS)
    public void testHashSet(Blackhole blackhole) {
        blackhole.consume(hashSet.contains(stringToFind));
    }

    @Benchmark

```

```

@BenchmarkMode (Mode.AverageTime)
@OutputTimeUnit (TimeUnit.NANOSECONDS)
public void testTreeSet (Blackhole blackhole) {
    blackhole.consume (treeSet.contains (stringToFind));
}
}

```

この`blackhole.consume()` えておいてください。でそれにります。また、ベンチマークのにはメインクラスがです。

```

package benchmark;

import org.openjdk.jmh.runner.Runner;
import org.openjdk.jmh.runner.RunnerException;
import org.openjdk.jmh.runner.options.Options;
import org.openjdk.jmh.runner.options.OptionsBuilder;

public class BenchmarkMain {
    public static void main (String[] args) throws RunnerException {
        final Options options = new OptionsBuilder()
            .include (CollectionFinderBenchmarkTest.class.getSimpleName ())
            .forks (1)
            .build ();

        new Runner (options).run ();
    }
}

```

そして、々はすべてされています。 `mvn package` をするだけで `/target` フォルダに `benchmarks.jar` がされます、ベンチマークテストをするだけです

```
java -cp target/benchmarks.jar benchmark.BenchmarkMain
```

ウォームアップとのりしがわれた、がられます。

```

# Run complete. Total time: 00:01:21

Benchmark                                     Mode  Cnt   Score   Error  Units
CollectionFinderBenchmarkTest.testHashSet    avgt   20   9.940 ± 0.270 ns/op
CollectionFinderBenchmarkTest.testTreeSet    avgt   20  98.858 ± 13.743 ns/op

```

その`blackhole.consume()` についてあなたののがあなたのアプリケーションのをえないならば、javaはたぶんそれをするでしょう。だから、それをけるために、あなたのベンチマークメソッドをいくつかのをすか、`Blackhole` オブジェクトをとってそれをさせることができます。

あなたには、なベンチマークをくことについてののをつけることができます [アレクセイ Shipilëvのブログ](#)で、 [ヤコブ Jenkovのブログ](#) とJavaのパフォーマンスのブログで [1](#)、[2](#)を。

オンラインでベンチマークをむ <https://riptutorial.com/ja/java/topic/9514/ベンチマーク>

122: モジュール

- `java.xml`がです。
- `public java.xml`がです。 するためにモジュールをにする
- `export com.example.foo;` はこのパッケージでパブリックタイプをできます
- `com.example.foo.impl`を`com.example.bar`にエクスポートします。 モジュールへのをする

モジュールのはされていますが、ではありません。これにより、のコードをJava 9でききできます。また、モジュールコードへのものです。

モジュールされたコードは、コンパイルにのモジュールにられます。これは、のすべてのモジュールのタイプをできるなモジュールですが、`exports`をつパッケージからのみできます。

のないモジュールのすべてのパッケージはにエクスポートされます。

キーワード `module` などは、モジュールでのがされていますが、ききのでとしてできます。

Examples

モジュールの

モジュールは`module-info.java`というのファイルにされ、モジュールとばれます。ソースコードのルートにするがあります。

```
|-- module-info.java
|-- com
    |-- example
        |-- foo
            |-- Foo.java
        |-- bar
            |-- Bar.java
```

ここにはなモジュールがあります

```
module com.example {
    requires java.httpclient;
    exports com.example.foo;
}
```

モジュールはでなければなりません。また、パッケージでされているのと同じ[Reverse-DNS](#)をすることをおめします。

Javaのクラスをむモジュールの`java.base`、どのモジュールでもにであり、インクルードするはありません。

`requires` モジュールのでは、たちがのモジュールをすることができ `java.httpclient` インポートされ

ます。

モジュールは、どのパッケージを `exports` するかをすることもできます。そのため、のモジュールにもされます。

`exports` でされたパッケージ `com.example.foo` は、のモジュールからえます。 `com.example.foo` サブパッケージはエクスポートされません。の `export` からです。

に、 `exports` にリストされていない `com.example.bar` はのモジュールにはされません。

オンラインでモジュールをむ <https://riptutorial.com/ja/java/topic/5286/モジュール>

123: ユニットテスト

き

ユニットテストは、テストのなであり、なアプリケーションをするためのなです。Javaでは、ユニットテストはほとんどの、のドキュメンテーションタグをつライブラリとフレームワークをしてされます。このスタブは、なツールとそれぞれのドキュメントにをなるとしてちます。

ユニットテストフレームワーク

Javaでのテストには、のフレームワークがされています。これまでにものあるオプションはJUnitです。それはのにされています

JUnit

JUnit4 - JUnit4ののためのされたタグ。まだされていません。

のテスト・フレームワークもし、なドキュメントがあります。

TestNG

ユニットテストツール

ユニットテストにはにもいくつかのツールがあります

Mockito - Mockingフレームワーク。オブジェクトをすることができます。のユニットののををリンクしないように、ののテストののなをするために。

JBehave - BDDフレームワークテストをユーザーのビヘイビアにリンクすることができます/シナリオのが。のなはありません。ここにリンクがあります。

Examples

ユニットテストとはですか

これはしのプライマです。スタブのとしてされていて、ドキュメンテーションにはがあるため、ほとんどののそれがあります。ユニットテストののをすでになっているは、なフレームワークがされているをみばしてください。

ユニットテストは、のモジュールがどおりにすることをなすることです。アプリケーションでは、でのモジュールのなをすることは、アプリケーションのをするためのなです。

のなをえてみましょう。

```
public class Example {
    public static void main (String args[]) {
        new Example();
    }

    // Application-level test.
    public Example() {
        Consumer c = new Consumer();
        System.out.println("VALUE = " + c.getVal());
    }

    // Your Module.
    class Consumer {
        private Capitalizer c;

        public Consumer() {
            c = new Capitalizer();
        }

        public String getVal() {
            return c.getVal();
        }
    }

    // Another team's module.
    class Capitalizer {
        private DataReader dr;

        public Capitalizer() {
            dr = new DataReader();
        }

        public String getVal() {
            return dr.readVal().toUpperCase();
        }
    }

    // Another team's module.
    class DataReader {
        public String readVal() {
            // Refers to a file somewhere in your application deployment, or
            // perhaps retrieved over a deployment-specific network.
            File f;
            String s = "data";
            // ... Read data from f into s ...
            return s;
        }
    }
}
```

したがって、このはです。 `DataReader` はファイルからデータをして `Capitalizer` し、すべてののをに
してから `Consumer` します。しかし、 `DataReader` はアプリケーションにきくしているため、テストリ
リースをするがうまで、このチェーンのテストをします。

さて、どこかのリリースでは、`toUpperCase()` によって、 `getVal()` のメソッド `Capitalizer` につてか
ら `toUpperCase()` にを `toLowerCase()`

```
// Another team's module.
class Capitalizer {
    ...

    public String getVal() {
        return dr.readVal().toLowerCase();
    }
}
```

らかに、これはされるをります。しかし、`DataReader`にうなプロセスのため、のテストまではかないでしょう。だから、このバグがシステムにあって//がぎると、プロダクトマネージャーがこれをして、`Consumer`けられたチームリーダーにあなたをかわせます。"なぜこれがこっているのですかあなたはをえましたか"らかに、あなたはです。あなたはがこっているのかかりません。あなたはこれにれるべきコードをしなかった。どうしてれたの

のところ、チームとコラボレーションの、がされ、がされます。しかし、それはをく。どのようにこれをぐことができましたか

らかに2つのことがあります。

テストをするがある

によるテストにしているため、このバグはあまりにもくかれませんでした。バグがにされるプロセスをするがです。から5ではありません。から5ではありません。から5ではありません。たった。

このでは、され、かれていないになバグを1つしていることをしておくがあります。アプリケーションでは、のモジュールがえずされているため、これらのモジュールはいたるところでびるがあります。1つのモジュールでかをするだけで、あなたが""されたるいがらかのでもにももされていることをすることができます。

ながなければ、システムにびむことになります。これがにされれば、をしてそしてをするなどながになるがあり、がわるにつれてがにりのでするがあります。このようなになることはましくありません。

テストはきめかくうがあります

のでした2のは、バグをするのにかったです。テスターがそれをついたときにプロダクトマネージャーがpingし、あなたがして、`Capitalizer`がいデータをしていたことをり、あなたのを`Capitalizer`チームにpingし、しました。

このなのとしさについてがでったのとじがここにあります。らかに、Javaにであればでも、されたをすばやくつけることができます。しかし、くの、をしてすることはずっとです。たぶん、`Capitalizer`チームがあなたにソースのないJARをしてくれたかもしれません。おそらくはのにしており、はにられていますたぶん1に1されるメール。にかかってしまうバグがするがあります

また、あるリリースではいくつかのバグがあります。

これをするために、なりなテストをにいたいとえていますモジュールがにすることをするための
いテストもですが、ここではそれがではありません。きのすべてのがどのようにするかをし、
そのをテストするがあります。

テストにる

々はにことをし、テストをとっていたをし`getVal()`のメソッド`Capitalizer`されたのをしました。さ
らに、コードをコミットするにテストがされたとします。システムにされたバグつまり、
`toUpperCase()`にきえられて`toLowerCase()`のバグがシステムにされることはありませんになるため
もをこさないだろう。たちはそれをテストでえ、はいをし、したをどのようにするかについての
のがられるでしょう。

これらのテストのについてはここではいくつかのけがありますが、フレームワークのドキュメン
トにリンクされていますにされています。うまくいけば、これはテストがなぜなののかのとしてち
ます。

オンラインでユニットテストをむ <https://riptutorial.com/ja/java/topic/8155/ユニットテスト>

124: ラムダ

き

ラムダは、をしてメソッドのインターフェイスをするためのかつなをします。それらをする、およびしなければならないコードのをらすことができます。クラスとていますが、はそれではありません。がです。

メソッドは、ではなくのメソッドをしてインタフェースをします。らはラムダにもしています。

- `-> {をす;}` //をすためののゼロアリティ。
- `-> //` の。のセミコロンはありません。
- `-> {function-body}` //をするラムダの。
- `parameterName -> //` 1つのラムダ。が1つしかないラムダでは、かっこはできます。
- `type parameterName、Type secondParameterName、... -> //` にリストされたパラメータをつをする `lambda`
- `parameterName、secondParameterName、... -> //` パラメータのパラメータをする。されたパラメータリストのサイズがされるインタフェースのサイズのうちの1つそして1つだけとするコンパイラによってできるコンテキストでのみできます。

Examples

ラムダをとってコレクションをべえる

ソートリスト

Java 8では、リストをソートするときに、またはきクラスで `java.util.Comparator` インタフェースをするがありました¹

Java SE 1.2

```
List<Person> people = ...
Collections.sort(
    people,
    new Comparator<Person>() {
        public int compare(Person p1, Person p2){
            return p1.getFirstName().compareTo(p2.getFirstName());
        }
    }
);
```

Java 8、クラスはラムダにきえることができます。コンパイラがにするので、パラメータ `p1` と `p2` のを `p2` することができます。

```

Collections.sort(
    people,
    (p1, p2) -> p1.getFirstName().compareTo(p2.getFirstName())
);

```

これは、`Comparator.comparing`と`::`ダブルコロンをしてされたメソッドをすることでできます。

```

Collections.sort(
    people,
    Comparator.comparing(Person::getFirstName)
);

```

インポートをすると、これをよりにすることができますが、ながするかどうかはのがあります。

```

import static java.util.Collections.sort;
import static java.util.Comparator.comparing;
//...
sort(people, comparing(Person::getFirstName));

```

このようにされたコンパレータは、させることもできます。たとえば、`をでした`、`じのがするは`、`thenComparing`メソッドもとします。

```

sort(people, comparing(Person::getFirstName).thenComparing(Person::getLastName));

```

1 - `Collections.sort...`は、`List` サブタイプであるコレクションにしてのみすることにしてください。 `Set` APIと `Collection` APIはのけをしません。

マップのべえ

ので`HashMap`のエントリをでソートすることができます。 `LinkedHashMap`ターゲットとしてするがあることにしてください。の`HashMap`のキーはけされていません。

```

Map<String, Integer> map = new HashMap(); // ... or any other Map class
// populate the map
map = map.entrySet()
    .stream()
    .sorted(Map.Entry.<String, Integer>comparingByValue())
    .collect(Collectors.toMap(k -> k.getKey(), v -> v.getValue(),
                              (k, v) -> k, LinkedHashMap::new));

```

Java ラムダ

インタフェース

Lambdasはなインタフェースでしかできません。これは1つのメソッドをつインタフェースです。インタフェースは、の`default`または`static`メソッドをつことができます。このため、`Single`

Abstract Method InterfacesまたはSAM Interfacesと呼ばれることもあります。

```
interface Foo1 {
    void bar();
}

interface Foo2 {
    int bar(boolean baz);
}

interface Foo3 {
    String bar(Object baz, int mink);
}

interface Foo4 {
    default String bar() { // default so not counted
        return "baz";
    }
    void quux();
}
```

なインタフェースをするときに、`@FunctionalInterface` `FunctionalInterface` アノテーションをすることができます。これにはなはありませんが、このアノテーションがしていないインタフェースにされているは、コンパイラエラーがされ、インタフェースをしないことをするをたします。

```
@FunctionalInterface
interface Foo5 {
    void bar();
}

@FunctionalInterface
interface BlankFoo1 extends Foo3 { // inherits abstract method from Foo3
}

@FunctionalInterface
interface Foo6 {
    void bar();
    boolean equals(Object obj); // overrides one of Object's method so not counted
}
```

に、これはのメソッドをとっているため、これはインタフェースではありません。

```
interface BadFoo {
    void bar();
    void quux(); // <-- Second method prevents lambda: which one should
                // be considered as lambda?
}
```

これはメソッドをたないため、なインターフェースでもありません。

```
interface BlankFoo2 { }
```

のことにしてください。あなたがっていると

```
interface Parent { public int parentMethod(); }
```

そして

```
interface Child extends Parent { public int ChildMethod(); }
```

それで、2つのされたメソッドがあるため、Child はなインターフェースにはなりません。

Java 8はまた、パッケージ `java.util.function` にのテンプレートインターフェースをしています。たとえば、みみのインターフェース `Predicate<T>` は、`T` をして `boolean` をすのメソッドをラップします。

ラムダ

ラムダのはのとおりでです。

```
FunctionalInterface fi = () -> System.out.println("Hello");
```

`fi` は `FunctionalInterface` をするクラスにたクラスのシングルトンインスタンスをし、そのメソッドのは `{ System.out.println("Hello"); }`。いえれば、はほほのものとです。

```
FunctionalInterface fi = new FunctionalInterface() {  
    @Override  
    public void theOneMethod() {  
        System.out.println("Hello");  
    }  
};
```

ラムダでは、`this`、`super` または `toString()` ようなのは、しくされたオブジェクトではなく、りてがわれるクラスをするため、ラムダはクラスと "ほとんど" です。

のインターフェースにはメソッドが1つしかなくてはならないため、ラムダをするときにメソッドのをすることはできませんが、そうするはありません。

ラムダのがでないえは、オーバーロードされたメソッド、ラムダにキャストをして、コンパイラにそのがどうあるべきかをえることができます

```
Object fooHolder = (Fool) () -> System.out.println("Hello");  
System.out.println(fooHolder instanceof Fool); // returns true
```

インターフェースのメソッドがパラメータをとる、これらのローカルはラムダののにれます。パラメータのをするはなく、インターフェースからってくるのですもありますただし、にじてパラメータのをするのはエラーではありません。したがって、これらの2つのはです。

```
Foo2 longFoo = new Foo2() {
    @Override
    public int bar(boolean baz) {
        return baz ? 1 : 0;
    }
};
Foo2 shortFoo = (x) -> { return x ? 1 : 0; };
```

のが1つしかないは、ののかっこはできます。

```
Foo2 np = x -> { return x ? 1 : 0; }; // okay
Foo3 np2 = x, y -> x.toString() + y // not okay
```

の

ラムダのにされたコードがステートメントではなくJavaである、それはのをすメソッドとしてわれます。したがって、の2つはです。

```
UnaryOperator addOneShort = (x) -> (x + 1);
UnaryOperator addOneLong = (x) -> { return (x + 1); };
```

ローカルへのアクセスバリエーション

ラムダはクラスのなので、みスコープのローカルにアクセスするのとじにいます。はラムダでfinalものとしてわれ、されてはならない。

```
UnaryOperator makeAdder(int amount) {
    return (x) -> (x + amount); // Legal even though amount will go out of scope
    // because amount is not modified
}

UnaryOperator makeAccumulator(int value) {
    return (x) -> { value += x; return value; }; // Will not compile
}
```

このようにするをラップするがあるは、のコピーをするのオブジェクトをするがあります。ラムダをしたJava Closureでしくしています。

ラムダけれ

ラムダはインターフェイスのなので、メソッドがラムダをけれるためになをするはありません。インターフェイスをとるはラムダをけれることもできます。

```
public void passMeALambda(Foo1 f) {
    f.bar();
}
passMeALambda(() -> System.out.println("Lambda called"));
```

ラムダの

ラムダはそれではのをたない。りのとともにパラメータのとがいくつかのをえることはですが、そのようなはどのにりてることができるかをします。ラムダは、のいずれかので、なインターフェイスタイプにりてられたときにタイプをけります。

- えば `myPredicate = s -> s.isEmpty()` ように、への
- をつパラメータとしてします。たとえば、 `stream.filter(s -> s.isEmpty())`
- をすからす。えば、 `return s -> s.isEmpty()`
- それをにキャストする、えば `(Predicate<String>) s -> s.isEmpty()`

タイプへのそのようなりてがなされるまで、ラムダはなタイプをさない。として、ラムダ。 `-> o.isEmpty()` えてみ `o -> o.isEmpty()`。じラムダをくのものにりてることができます。

```
Predicate<String> javaStringPred = o -> o.isEmpty();
Function<String, Boolean> javaFunc = o -> o.isEmpty();
Predicate<List> javaListPred = o -> o.isEmpty();
Consumer<String> javaStringConsumer = o -> o.isEmpty(); // return value is ignored!
com.google.common.base.Predicate<String> guavaPredicate = o -> o.isEmpty();
```

ラムダはじようにえますが、いにりてることはできませんが、ここでしたはになるのものです。

メソッド

メソッドでは、ラムダのわりにとしてすことができる、のあるインターフェイスにする、あらかじめされたメソッドまたはインスタンスメソッドをできます。

モデルがあるとします。

```
class Person {
    private final String name;
    private final String surname;

    public Person(String name, String surname){
        this.name = name;
        this.surname = surname;
    }

    public String getName(){ return name; }
    public String getSurname(){ return surname; }
}

List<Person> people = getSomePeople();
```

インスタンスメソッドのインスタンスへ

```
people.stream().map(Person::getName)
```

ラムダ

```
people.stream().map(person -> person.getName())
```

ここでは、タイプ `Person` インスタンス・メソッド `getName()` へのメソッドがされています。コレクションであることがわかっているので、インスタンスのメソッドでかゝるがびされます。

インスタンスメソッドのインスタンスへの

```
people.forEach(System.out::println);
```

`System.out` は `PrintStream` インスタンスなので、こののインスタンスへのメソッドがとしてされます。

ラムダ

```
people.forEach(person -> System.out.println(person));
```

メソッド

ストリームをするは、メソッドへのをすることもできます。

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);  
numbers.stream().map(String::valueOf)
```

ここでは、`String` の `static` `valueOf()` メソッドへのをします。したがって、コレクションのインスタンスオブジェクトは、`valueOf()` としてされます。

ラムダ

```
numbers.stream().map(num -> String.valueOf(num))
```

コンストラクタへの

```
List<String> strings = Arrays.asList("1", "2", "3");  
strings.stream().map(Integer::new)
```

コレクションのをするについては、「[ストリームのをコレクションにみむ](#)」をしてください。

`Integer`ののStringコンストラクタがここでされ、としてされたをしてをしします。この、がをすり、ストリームはにマップされます。ラムダ

```
strings.stream().map(s -> new Integer(s));
```

カンニングペーパー

メソッド	コード	ラムダ
メソッド	<code>TypeName::method</code>	<code>(args) -> TypeName.method(args)</code>
メソッドインスタンス [*] の	<code>instance::method</code>	<code>(args) -> instance.method(args)</code>
メソッドインスタンスなし	<code>TypeName::method</code>	<code>(instance, args) -> instance.method(args)</code>
コンストラクタ ^{**}	<code>TypeName::new</code>	<code>(args) -> new TypeName(args)</code>
コンストラクタ	<code>TypeName[]::new</code>	<code>(int size) -> new TypeName[size]</code>

^{*} `instance`は、インスタンスへのをするのです。例えば、`getInstance()::method`、`this::method`

^{**} `TypeName`がなクラスである、コンストラクタはクラスインスタンスのスコープでのみです

のインタフェースの

ラムダをのインターフェイスでしたいがあります。これは、マッパーインタフェース [java.io.Serializable](#)などでは、メソッドをししないため、ほとんどのです。

たとえば、カスタムComparator [TreeSet](#)をし、シリアルしてネットワークでしたいとします。なアプローチ

```
TreeSet<Long> ts = new TreeSet<>((x, y) -> Long.compare(y, x));
```

コンパレータのラムダは`Serializable`していないのでしません。タイプをし、このラムダをシリアルイズにするがあることをにすることで、これをできます。

```
TreeSet<Long> ts = new TreeSet<>(  
    (Comparator<Long> & Serializable) (x, y) -> Long.compare(y, x));
```

ほとんどの、シリアルがな[Apache Spark](#)などのフレームワークをしているなど、にタイプをしているは、のインターフェイスをして、わりにコードでできます。

```
public interface SerializableComparator extends Comparator<Long>, Serializable {}

public class CustomTreeSet {
    public CustomTreeSet(SerializableComparator comparator) {}
}
```

これにより、されたコンパレータがであることがされます。

ラムダとパターン

なシナリオでは、lambdaをFunctionalInterfaceとしてするいくつがあります。ラムダによってできるかなりのユースケースは、いわゆるExecute-Aroundパターンです。このパターンでは、ユースケースのコードをりくのシナリオでとなるのセットアップ/ティアダウンコードがされています。これのなは、ファイルio、データベースio、try / catchブロックです。

```
interface DataProcessor {
    void process( Connection connection ) throws SQLException;;
}

public void doProcessing( DataProcessor processor ) throws SQLException{
    try (Connection connection = DBUtil.getDatabaseConnection();) {
        processor.process(connection);
        connection.commit();
    }
}
```

ラムダでこのメソッドをびすと、のようになります。

```
public static void updateMyDAO(MyVO vo) throws DatabaseException {
    doProcessing((Connection conn) -> MyDAO.update(conn, ObjectMapper.map(vo)));
}
```

これはI/Oにされません。これは、の/タスクがさなでなあらゆるシナリオにできます。このパターンのなは、コードのとDRYりしはしないでくださいをすることです。

ラムダをのインターフェイスでする

Lambdaは、メソッドインタフェースのインラインインプリメンテーションコードと、のでってきたようにそれらをすをすることをしてしています。々はそれらをインタフェースとぶ。

たとえば、クラスにRunnableをし、スレッドをすると、のようになります。

```
//Old way
new Thread(
    new Runnable(){
        public void run(){
            System.out.println("run logic...");
        }
    }
).start();
```

```
//lambdas, from Java 8
new Thread(
    ()-> System.out.println("run logic...")
).start();
```

のように、カスタムインターフェースがあるとしましょう

```
interface TwoArgInterface {
    int operate(int a, int b);
}
```

どのようにラムダをしてコードにこのインターフェースのをしますかのRunnableのとじです。のドライバプログラムをしてください

```
public class CustomLambda {
    public static void main(String[] args) {

        TwoArgInterface plusOperation = (a, b) -> a + b;
        TwoArgInterface divideOperation = (a,b)->{
            if (b==0) throw new IllegalArgumentException("Divisor can not be 0");
            return a/b;
        };

        System.out.println("Plus operation of 3 and 5 is: " + plusOperation.operate(3, 5));
        System.out.println("Divide operation 50 by 25 is: " + divideOperation.operate(50,
25));

    }
}
```

、**return**はのメソッドではなく、ラムダからのみります。

returnメソッドは、のメソッドではなくラムダからのみされます。

これはScalaとKotlinとはなることにしてください

```
void threeTimes(IntConsumer r) {
    for (int i = 0; i < 3; i++) {
        r.accept(i);
    }
}

void demo() {
    threeTimes(i -> {
        System.out.println(i);
        return; // Return from lambda to threeTimes only!
    });
}
```

などのみみのように、のをしようとするとき、これはしないにつながるforのループがreturnがなります

```
void demo2() {
```



```

for (int i = 0; i < 3; i++) {
    System.out.println(i);
    return; // Return from 'demo2' entirely
}
}

```

ScalaとKotlinでは、`demo`と`demo demo2`はとも0ます。しかしこれはしていません。Javaのアプローチは、リファクタリングとクラスのととしています。つまり、コードのの`return`であり、のコードはじようにします。

```

void demo3() {
    threeTimes(new MyIntConsumer());
}

class MyIntConsumer implements IntConsumer {
    public void accept(int i) {
        System.out.println(i);
        return;
    }
}

```

したがって、Javaの`return`はクラスメソッドとリファクタリングとよりしますが、みみの`for`や`while`では、これらはなまます。

このため、の2つはJavaです。

```

IntStream.range(1, 4)
    .map(x -> x * x)
    .forEach(System.out::println);
IntStream.range(1, 4)
    .map(x -> { return x * x; })
    .forEach(System.out::println);

```

さらに、Javaでは`try-with-resource`のがです。

```

class Resource implements AutoCloseable {
    public void close() { System.out.println("close()"); }
}

void executeAround(Consumer<Resource> f) {
    try (Resource r = new Resource()) {
        System.out.print("before ");
        f.accept(r);
        System.out.print("after ");
    }
}

void demo4() {
    executeAround(r -> {
        System.out.print("accept() ");
        return; // Does not return from demo4, but frees the resource.
    });
}

```

`before accept()` after `close()` にされます。ScalaとKotlinのセマンティクスでは、`try-with-resources`はじられませんが、`before accept()`のみされます。

ラムダによるJavaクローズ

ラムダがむスコープグローバルまたはローカルのをすると、ラムダクロージャがされます。これをうためのルールは、インラインメソッドとクラスのルールとじです。

ラムダでされるみスコープのローカルは、`final`なければなりません。Java 8ラムダをサポートするものバージョンでは、コンテキストで`final`するはありませんが、そのようにわなければなりません。えば

```
int n = 0; // With Java 8 there is no need to explicit final
Runnable r = () -> { // Using lambda
    int i = n;
    // do something
};
```

これは、`n`のがされないうりです。ラムダのまたはのをしようとする、のコンパイルエラーがします。

"ラムダからされるローカルは、またはにでなければならぬ"。

えば

```
int n = 0;
Runnable r = () -> { // Using lambda
    int i = n;
    // do something
};
n++; // Will generate an error.
```

ラムダでするをするがある、のはの`final`コピーをしてそのコピーをすることです。えば

```
int n = 0;
final int k = n; // With Java 8 there is no need to explicit final
Runnable r = () -> { // Using lambda
    int i = k;
    // do something
};
n++; // Now will not generate an error
r.run(); // Will run with i = 0 because k was 0 when the lambda was created
```

もちろん、ラムダにはのへのがされません。

Javaはのクロージャをサポートしていないことにしてください。Javaラムダは、インスタンスされたのことができることはできません。をしたりしたりするクロージャをするは、のクラスをしてシミュレートするがあります。えば

```
// Does not compile ...
```

```
public IntUnaryOperator createAccumulator() {
    int value = 0;
    IntUnaryOperator accumulate = (x) -> { value += x; return value; };
    return accumulate;
}
```

のは、のでコンパイルされません。のようにコンパイルエラーをすることができます

```
// Compiles, but is incorrect ...
public class AccumulatorGenerator {
    private int value = 0;

    public IntUnaryOperator createAccumulator() {
        IntUnaryOperator accumulate = (x) -> { value += x; return value; };
        return accumulate;
    }
}
```

は、インスタンスがし、ステートレスでなければならないことをす `IntUnaryOperator` インターフェイスのが `IntUnaryOperator` れて `IntUnaryOperator` ことです。そのようなクロージャがオブジェクトをけれるみみにされた、クラッシュやのとなりがちです。なをカプセルするクロージャは、のクラスとしてするがあります。えば。

```
// Correct ...
public class Accumulator {
    private int value = 0;

    public int accumulate(int x) {
        value += x;
        return value;
    }
}
```

ラムダ・リスナーの

クラスのリスナー

Java 8よりでは、のコードにすように、クラスをして `JButton` のクリックイベントをすることはにです。このでは、 `btn.addActionListener` のスコープにリスナーをするをします。

```
JButton btn = new JButton("My Button");
btn.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button was pressed");
    }
});
```

ラムダリスナー

`ActionListener` インタフェースは `actionPerformed()` メソッドを1つしかしていないため、のコードをきえるためにラムダをするがあることをするなインタフェースです。のは、のようにラムダをっ

てきすことができます

```
JButton btn = new JButton("My Button");
btn.addActionListener(e -> {
    System.out.println("Button was pressed");
});
```

なスタイルからラムダスタイルまで

な

```
interface MathOperation{
    boolean unaryOperation(int num);
}

public class LambdaTry {
    public static void main(String[] args) {
        MathOperation isEven = new MathOperation() {
            @Override
            public boolean unaryOperation(int num) {
                return num%2 == 0;
            }
        };

        System.out.println(isEven.unaryOperation(25));
        System.out.println(isEven.unaryOperation(20));
    }
}
```

ラムダスタイル

1. クラスとインタフェースをします。

```
public class LambdaTry {
    public static void main(String[] args) {
        MathOperation isEven = (int num) -> {
            return num%2 == 0;
        };

        System.out.println(isEven.unaryOperation(25));
        System.out.println(isEven.unaryOperation(20));
    }
}
```

2. オプションの

```
MathOperation isEven = (num) -> {
    return num%2 == 0;
};
```

3. パラメータのは、パラメータのりのオプション

```
MathOperation isEven = num -> {
```

```
return num%2 == 0;
};
```

4. オプションのカッコに1しかない
5. オプションのreturnキーワード。に1しかない

```
MathOperation isEven = num -> num%2 == 0;
```

ラムダとメモリ

Javaラムダはクロージャであるため、むれキシカルスコープのものを「」できます。すべてのラムダがかをキャプチャしているわけではありませんが、`s -> s.length()`のようなラムダはもせず、ステートレスキャプチャラムダとばれ、キャプチャされたをするためにオブジェクトがです。このコードスニペットでは、`lambda () -> j`はキャプチャラムダであり、オブジェクトがされるときにオブジェクトがりてられるがあります。

```
public static void main(String[] args) throws Exception {
    for (int i = 0; i < 1000000000; i++) {
        int j = i;
        doSomethingWithLambda(() -> j);
    }
}
```

`new` キーワードがスニペットのどこにもれないのですぐにはわからないかもしれませんが、このコードでは、`() -> j`ラムダのインスタンスをすために1,000,000,000の々のオブジェクトをしなければなりません。ただし、Java 1⁰のバージョンでは、にラムダインスタンスがされたり、のでされたりするように、これをできるがあります。

たとえば、Java 9では、Javaビルドシーケンスにオプションの「リンク」フェーズがされており、このようなグローバルなをうがされます。

ラムダとをってリストからのをする

Java 8からは、ラムダとをできます。

ラムダとaをして、リストからのをします。このでは、18であればすべてのがそのをプリントアウトされます。

クラス

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

```

    public int getAge() { return age; }
    public String getName() { return name; }
}

```

`java.util.function.Predicate` パッケージからのみみインターフェース `Predicate` は、 `boolean test(T t)` メソッドをつインターフェースです。

```

import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class LambdaExample {
    public static void main(String[] args) {
        List<Person> personList = new ArrayList<Person>();
        personList.add(new Person("Jeroen", 20));
        personList.add(new Person("Jack", 5));
        personList.add(new Person("Lisa", 19));

        print(personList, p -> p.getAge() >= 18);
    }

    private static void print(List<Person> personList, Predicate<Person> checker) {
        for (Person person : personList) {
            if (checker.test(person)) {
                System.out.print(person + " matches your expression.");
            } else {
                System.out.println(person + " doesn't match your expression.");
            }
        }
    }
}

```

`print(personList, p -> p.getAge() >= 18);` メソッドはラムダをとります `Predicate` はパラメータとしてされるため。なをできます。チェッカーのテストメソッドは、このがしいかどうかをチェックします `checker.test(person)`。

例えば、 `print(personList, p -> p.getName().startsWith("J"));` ように、これをのものににすることができます `print(personList, p -> p.getName().startsWith("J"));`。これは、のが「J」でまるかどうかをチェックします。

オンラインでラムダをむ <https://riptutorial.com/ja/java/topic/91/ラムダ>

125: リスト

き

リストはけられたのです。Javaでは、リストは[Java Collections Framework](#)のです。リストは、[java.util.List](#)インタフェース、[java.util.Collection](#)。

- `ls.addE; //`をする
- `ls.removeE; //`をする
- `forE elementls{} //`をりします
- `ls.toArray`しい`[ls.length];` //ストリングのリストをストリングのにする
- `ls.getint index; //`されたインデックスにあるをします。
- `ls.setint index、 E element; //`されたにをきえます。
- `ls.isEmpty; //`にがないは`true`、そうでないは`false`をします。
- `ls.indexOf`オブジェクト`o; //`された`o`ののインデックスをします。しないは`-1`をします。
- `ls.lastIndexOf`オブジェクト`o; //`された`o`ののインデックスをします。しないは`-1`をします。
- `ls.size; //`リストののをします。

[リスト](#)は、けられたのをするオブジェクトです。「オーダーされた」とは、がのでされていることをします。1つはに、もう1つは2にきます。々のはに「`」`とばれます。Javaのリストには、`、`のがあります。

- リストには`0`のがまれています。
- リストにしたがまれているがあります。つまり、をリストにすることができます。
- リストはをのでします。つまり、にがる、がるなどをします。
- には、リストのをすインデックスがあります。のはインデックス`0`をち、のはインデックス`1`をちます。
- リストは、リストの、`、`またはのインデックスにをすることをします。
- リストにのがまれているかどうかをテストするとは、`、`リストのをべることをします。これは、このチェックをするがリストのサイズにする[On](#)であることをします。

のでリストにをすると、のがすべて「に」または「に」します。すれば、`n`のをするように、インデックス`N + 1`とする`n`になるようにされるをさせます。えは

```
List<String> list = new ArrayList<>();
list.add("world");
System.out.println(list.indexOf("world"));           // Prints "0"
// Inserting a new value at index 0 moves "world" to index 1
list.add(0, "Hello");
System.out.println(list.indexOf("world"));           // Prints "1"
System.out.println(list.indexOf("Hello"));           // Prints "0"
```

Examples

ジェネリックリストのソート

`Collections` クラスは、リストをソートする2つのメソッドをします。

- `sort(List<T> list)` は、 `T extends Comparable<? super T>` どこにある `T extends Comparable<? super T>` をすリストに `T extends Comparable<? super T>` 、
- ののリストに `sort(List<T> list, Comparator<? super T> c)`

をするには、ソートされるリストのクラスをするがありますが、これはずしもではありません。また、デフォルトソートや、のでのソートオーダー、ソートなどのタスクがながありますが、ましくないもあります。

のクラスのインスタンスであるオブジェクトをソートするタスクがあるとします。

```
public class User {
    public final Long id;
    public final String username;

    public User(Long id, String username) {
        this.id = id;
        this.username = username;
    }

    @Override
    public String toString() {
        return String.format("%s:%d", username, id);
    }
}
```

`Collections.sort(List<User> list)` を `User` するには、 `Comparable` インターフェースをするために `User` クラスをするがあります。えは

```
public class User implements Comparable<User> {
    public final Long id;
    public final String username;

    public User(Long id, String username) {
        this.id = id;
        this.username = username;
    }

    @Override
    public String toString() {
        return String.format("%s:%d", username, id);
    }

    @Override
    /** The natural ordering for 'User' objects is by the 'id' field. */
    public int compareTo(User o) {
        return id.compareTo(o.id);
    }
}
```



```
}
```

`String`、`Long`、`Integer`などの多くのJavaクラスは`Comparable`インターフェイスをしているため、これらのリストがデフォルトでソートになり、のクラスの`compare`または`compareTo`がになります

のにより、クラスのけについて`User`オブジェクトのリストをにソートできます。この、`id`についてけすることがされています。え

```
List<User> users = Lists.newArrayList(  
    new User(33L, "A"),  
    new User(25L, "B"),  
    new User(28L, "C"));  
Collections.sort(users);  
  
System.out.print(users);  
// [B:25, C:28, A:33]
```

しかし、`User`オブジェクトを`id`ではなく`name`でソートしたいとします。あるいは、`Comparable`のためにクラスをできなかったとします。

これは、`Comparator`をした`sort`メソッドがなです。

```
Collections.sort(users, new Comparator<User>() {  
    @Override  
    /* Order two 'User' objects based on their names. */  
    public int compare(User left, User right) {  
        return left.username.compareTo(right.username);  
    }  
});  
System.out.print(users);  
// [A:33, B:25, C:28]
```

Java SE 8

Java 8では、クラスのわりにラムダをできます。は1ライナーにします

```
Collections.sort(users, (l, r) -> l.username.compareTo(r.username));
```

さらに、Java 8では、`List`インターフェースにデフォルトの`sort`メソッドがされているため、ソートがさらにになります。

```
users.sort((l, r) -> l.username.compareTo(r.username))
```

リストの

あなたのリストにタイプをえる

リストをするには、クラスなど、`String`がです。これはあなたの`List`のタイプです。`List`は、されたタイプのオブジェクトのみがされます。え

```
List<String> strings;
```

"string1"、 "hello world!"することができます "hello world!"、 "goodbye"などですが、 9.2はできません

```
List<Double> doubles;
```

9.2ですが、 "hello world!"はできません"hello world!"。

あなたのリストをする

のリストにかをしようとする、NullPointerExceptionがします。これは、 stringsとdoublesが nullにしいためです。

リストをするには、の2つのがあります。

オプション**1List**をするクラスをする

Listはインタフェースであり、コンストラクタをたず、むしろクラスがオーバーライドするがあるメソッドをします。 ArrayListはもにされるList ArrayListが、 LinkedListもです。そこで、のようリストをします。

```
List<String> strings = new ArrayList<String>();
```

または

```
List<String> strings = new LinkedList<String>();
```

Java SE 7

Java SE 7からは、 **ダイヤモンド**をできます。

```
List<String> strings = new ArrayList<>();
```

または

```
List<String> strings = new LinkedList<>();
```

オプション**2Collections**クラスをする

Collectionsクラスは、 ListなしでListをする2つのなメソッドをします

- emptyList() のリストをします。
- singletonList(T) singletonList(T)のリストをし、されたをします。

のListをしてデータをする

- addAll(L, T...)

されたすべてののを、のパラメータとしてされたリストにします。

```
import java.util.List;
import java.util.Collections;

List<Integer> l = Collections.emptyList();
List<Integer> l1 = Collections.singletonList(42);
Collections.addAll(l1, 1, 2, 3);
```

アクセス

List APIには、アクセスのための8つのメソッドがあります。

- add(T type)
- add(int index, T type)
- remove(Object o)
- remove(int index)
- get(int index)
- set(int index, E element)
- int indexOf(Object o)
- int lastIndexOf(Object o)

だから、リストがあれば

```
List<String> strings = new ArrayList<String>();
```

「Hello world」をしたかったのです。と "さようなら"それには、たちはそれをそうするでしょう

```
strings.add("Hello world!");
strings.add("Goodbye world!");
```

たちのリストには2つのがまれています。ここで「プログラムの」をしたいとします。リストのにたちはこれをのようになります

```
strings.add(0, "Program starting!");
```

のは0です。

さて、「さようならたちはこれをのようになすことができます

```
strings.remove("Goodbye world!");
```

そして、のしたいこのは「プログラム」となります、のようになすことができます。

```
strings.remove(0);
```

1. リストをおよびするとリストがされ、リストがにされている、
ConcurrentModificationExceptionがするがあります。

2. リストクラス、されるメソッド、およびリストの、またはにを/するかどうかによって、のとは $O(1)$ または $O(N)$ になります。

されたでリストのをするには、`E get(int index);` List APIのメソッド例えば

```
strings.get(0);
```

リストののをします。

`set(int index, E element);`をして、されたにあるのをきえることができ`set(int index, E element);`。
例えば

```
strings.set(0, "This is a replacement");
```

これにより、リストののとして "This is a replacement"がされます。

setメソッドは、0のをきします。0にしいStringをせず、いものを1にプッシュします。

`indexOf(Object o);`としてされたオブジェクトののをします。リストにオブジェクトがない、-1のがされます。ののきで

```
strings.indexOf("This is a replacement")
```

リストの0のに "これは"をしているので、0がされることがされます。リストにのオカレンスがある
`indexOf(Object o);`のようにのオカレンスのインデックスがされます。 `int
lastIndexOf(Object o)`をびすことで、リストののオカレンスのインデックスをできます。したがって、の「これはきえ」をした

```
strings.add("This is a replacement");  
strings.lastIndexOf("This is a replacement");
```

は1がされ、0はされません。

リストのをする

このでは、「hello」、「how」、「are」、「you」の4つのをむStringのリストがあります。

をするのは、for-eachループをすることです。

```
public void printEachElement(List<String> list){  
    for(String s : list){  
        System.out.println(s);  
    }  
}
```

どちらがされます

```
hello,  
how  
are  
you?
```

それらをすべてじにするには、StringBuilderをできます。

```
public void printAsLine(List<String> list){  
    StringBuilder builder = new StringBuilder();  
    for(String s : list){  
        builder.append(s);  
    }  
    System.out.println(builder.toString());  
}
```

されます

```
hello, how are you?
```

あるいは、のインデックス [ArrayListのiのインデックスにあるへのアクセスを](#)をしてリストをすることができます。これは、リンクされたリストではではありません。

リスト **A**にするリスト **B**からをする

あなたは2つのリストAとBをとっている、とあなたは**B**からあなたは、このにをとっているすべてのをしたいですとします

```
List.removeAll(Collection c);
```

```
public static void main(String[] args) {  
    List<Integer> numbersA = new ArrayList<>();  
    List<Integer> numbersB = new ArrayList<>();  
    numbersA.addAll(Arrays.asList(new Integer[] { 1, 3, 4, 7, 5, 2 }));  
    numbersB.addAll(Arrays.asList(new Integer[] { 13, 32, 533, 3, 4, 2 }));  
    System.out.println("A: " + numbersA);  
    System.out.println("B: " + numbersB);  
  
    numbersB.removeAll(numbersA);  
    System.out.println("B cleared: " + numbersB);  
}
```

これはされます

A[1、3、4、7、5、2]

B[13,32,533,3,4,2]

Bをクリア[13,32,533]

2つのリストの

AとBの2つのリストがあり、のリストにするをつけるがとします。

`List.retainAll()` メソッドをびすだけでそれをうことができます。

```
public static void main(String[] args) {
    List<Integer> numbersA = new ArrayList<>();
    List<Integer> numbersB = new ArrayList<>();
    numbersA.addAll(Arrays.asList(new Integer[] { 1, 3, 4, 7, 5, 2 }));
    numbersB.addAll(Arrays.asList(new Integer[] { 13, 32, 533, 3, 4, 2 }));

    System.out.println("A: " + numbersA);
    System.out.println("B: " + numbersB);
    List<Integer> numbersC = new ArrayList<>();
    numbersC.addAll(numbersA);
    numbersC.retainAll(numbersB);

    System.out.println("List A : " + numbersA);
    System.out.println("List B : " + numbersB);
    System.out.println("Common elements between A and B: " + numbersC);
}
```

のリストをのリストにする

```
List<Integer> nums = Arrays.asList(1, 2, 3);
List<String> strings = nums.stream()
    .map(Object::toString)
    .collect(Collectors.toList());
```

あれは

1. リストからストリームをする
2. `Object::toString` をしてをマップする
3. `Collectors.toList()` をして `String` を `List` する

ArrayListからの、、および

`ArrayList` は、Javaのみみデータの1つ `ArrayList` 。これは、オブジェクトをするためのにするのな
いデータのサイズです。

`AbstractList` クラスをし、 `List` インタフェースをしします。 `ArrayList` は、をするをめることができ
ます。 `ArrayList` クラスはしていないので、 `ArrayList` をしてをうはが `ArrayList` 。 `ArrayList` はイン
デックススペースであるため、ランダムアクセスが `ArrayList` 。 がリストからされたときににするシ
フトのため、 `ArrayList` がくなります。

`ArrayList` は、のようにできます。

```
List<T> myArrayList = new ArrayList<>();
```

T **Generics** は、 `ArrayList` にされる `ArrayList` 。

`ArrayList`のはのObjectにすることができます。はプリミティブではありませんわりにラッパークラスをしてください。

`ArrayList`にをするには、`add()`メソッドをします。

```
myArrayList.add(element);
```

またはのインデックスにアイテムをするには

```
myArrayList.add(index, element); //index of the element should be an int (starting from 0)
```

`ArrayList`からをするには、`remove()`メソッドをします。

```
myArrayList.remove(element);
```

またはのインデックスからアイテムをするには

```
myArrayList.remove(index); //index of the element should be an int (starting from 0)
```

Listのインプレース

このでは、きえがきえられるとじにあることをしながら`List`をきえるについてします。

これは、のをしてできます。

- `set` index、`T` type
- `indexOf` タイプ

"Program starting"、"Hello world"をむ`ArrayList`をえてみましょう。と"さようなら"

```
List<String> strings = new ArrayList<String>();
strings.add("Program starting!");
strings.add("Hello world!");
strings.add("Goodbye world!");
```

きえたいのインデックスがわかっているは、に`set`をののようにします。

```
strings.set(1, "Hi world");
```

インデックスがわからないは、にすることができます。えは

```
int pos = strings.indexOf("Goodbye world!");
if (pos >= 0) {
    strings.set(pos, "Goodbye cruel world!");
}
```

ノート

1. set では `ConcurrentModificationException` はしません。
2. set は `ArrayList` では $O(1)$ 、 `LinkedList` では $O(N)$ です。
3. `ArrayList` または `LinkedList` の `indexOf` が $O(N)$ 。

リストをにする

`Collections` クラスは、リストをにするをします。

```
List<String> ls = new ArrayList<String>();  
List<String> unmodifiableList = Collections.unmodifiableList(ls);
```

1つのアイテムをつなリストがなは、のものをできます。

```
List<String> unmodifiableList = Collections.singletonList("Only string in the list");
```

リストのオブジェクトをする

`Collections` クラスでは、さまざまなメソッドリストは `ls` をして、リストのオブジェクトをすることができます。

リストをする

```
Collections.reverse(ls);
```

リストの

`rotate` メソッドにはのがです。これは、にってそれをするためにいくつかのスポットです。そのをにします。

```
List<String> ls = new ArrayList<String>();  
ls.add(" how");  
ls.add(" are");  
ls.add(" you?");  
ls.add("hello,");  
Collections.rotate(ls, 1);  
  
for(String line : ls) System.out.print(line);  
System.out.println();
```

これは "こんにちは、"

リストのをシャッフルする

のじリストをして、リストのをシャッフルすることができます

```
Collections.shuffle(ls);
```

また、`java.util.Random` オブジェクトにオブジェクトをランダムにするためにすることもできます


```
Random random = new Random(12);
Collections.shuffle(ls, random);
```

リストをするクラス。と

[List](#) インタフェースは、なるクラスによってされています。それぞれには、さまざまでそれをし、さまざまをするのがあります。

リストをするクラス

`java.util.List` インタフェースをするJava SE 8のすべての`public`クラスです。

1. クラス
 - `AbstractList`
 - `AbstractSequentialList`
2. なクラス
 - リスト
 - `AttributeList`
 - `CopyOnWriteArrayList`
 - `LinkedList`
 - ロールリスト
 - `RoleUnresolvedList`
 - スタック
 - ベクター

のさのからのと

リスト

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

[ArrayList](#)は、`List`インタフェースのサイズなです。にリストをする`ArrayList`は、のサイズをするためのメソッド `List`インタフェースをするメソッドにえてをします。

Integerの**ArrayList**をサイズ**100**でします。

```
List<Integer> myList = new ArrayList<Integer>(100); // Constructs an empty list with the
specified initial capacity.
```

- PROS

size、isEmpty、**get**、**set**、iterator、およびlistIteratorオペレーションはO(1)にされます。したがって、リストの操作をするのにコストが低いです。

```
int e1 = myList.get(0); // \
int e2 = myList.get(10); // | => All the same constant cost => O(1)
myList.set(2,10); // /
```

-

ArrayListのサイズを増やすと、すべての要素を新しい配列にコピーする必要があるため、大きなコストがかかります。ただし、[ドキュメント](#)から

はわかります。つまり、n要素を増やすにはO(n)です。

を減らすには、O(1)です。

AttributeList

るに

CopyOnWriteArrayList

るに

LinkedList

```
public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

LinkedListは、[リンクされたリスト](#)によって、ノードと呼ばれるしてリンクされたレコードのセットでされるリンクされたデータによってされます。

の**LinkedList**をする

```
List<Integer> myList = new LinkedList<Integer>(); // Constructs an empty list.
```

- PROS

リストの追加や削除をするときには、O(1)があります。

```
myList.add(10); // \
```

```
myList.add(0,2); // | => constant time => O(1)
myList.remove(); // /
```

・めり [ドキュメント](#) から

リストにインデックスをけるオペレーションは、されたインデックスにいほうからリストのまたはをします。

のような

```
myList.get(10); // \
myList.add(11,25); // | => worst case done in O(n/2)
myList.set(15,35); // /
```

ロールリスト

るに

RoleUnresolvedList

るに

スタック

るに

ベクター

るに

オンラインでリストをむ <https://riptutorial.com/ja/java/topic/2989/リスト>

126: リスト SET

き

トップレベルのListとSetコレクションのいと、JavaでListをするタイミングとJavaでSetをするタイミングをする

Examples

リストセット

```
import java.util.ArrayList;
```

```
import java.util.HashSet; import java.util.List; import java.util.Set;
```

```
パブリッククラス SetAndListExample {public static void mainString [] args{System.out.println "List  
example ....."; リストリスト=しいArrayList; list.add "1"; list.add "2"; list.add "3"; list.add "4"; list.add  
"1";
```

```
    for (String temp : list){  
        System.out.println(temp);  
    }
```

```
    System.out.println("Set example .....");  
    Set<String> set = new HashSet<String>();  
    set.add("1");  
    set.add("2");  
    set.add("3");  
    set.add("4");  
    set.add("1");  
    set.add("2");  
    set.add("5");
```

```
    for (String temp : set){  
        System.out.println(temp);  
    }  
}
```

}

リストの..... 1 2 3 4 1..... 3 2 10 5 4

オンラインでリストSETをむ <https://riptutorial.com/ja/java/topic/10125/リストset>

127: リソース クラスパス

き

Javaでは、コンパイルされたクラスとにJARのにされているファイルベースのリソースをできます。このトピックでは、これらのリソースをロードし、コードでできるようにすることにをいています。

リソースとは、パスのようなファイルのようなデータで、クラスパスにします。リソースのものは、アプリケーションイメージ、サウンド、およびみりデータデフォルトなどをバンドルすることです。

リソースをとってアクセスすることができ[ClassLoader.getResource](#)と[ClassLoader.getResourceAsStream](#)。ものは、リソースをみるクラスとしパッケージにすることです。[Class.getResource](#)と[Class.getResourceAsStream](#)は、このなをしています。

`getResource`メソッドと[getResourceAsStream](#)メソッドののいは、がURLをし、がそのURLをき、`InputStream`をすことです。

`ClassLoader`のメソッドは、パスのようなリソースをとしてけり、そのにするエントリを[ClassLoader](#)のクラスパスののです。

- クラスパスのが.jarファイルの、されたの.jarエントリはとみなされます。
- クラスパスのがディレクトリの、されたをつそのディレクトリののファイルはとなされます。

リソースは、URLのパスにています。すべてのプラットフォームで、ディレクトリりとしてスラッシュ / をします。スラッシュでめることはできません。

`Class`のするメソッドは、をいてです。

- リソースはスラッシュでまり、そののはのスラッシュがされ、りのは[ClassLoader](#)のするメソッドにされます。
- リソースがスラッシュでまらないは、`getResource`メソッドまたは[getResourceAsStream](#)メソッドがびされているクラスにしてなとしてわれます。のリソースは、 パッケージは、 パッケージのであるパッケージ / となるクラスはスラッシュできえで、 がメソッドにさされます。

えば

```
package com.example;

public class ExampleApplication {
    public void readImage()
        throws IOException {
```

```

URL imageURL = ExampleApplication.class.getResource("icon.png");

// The above statement is identical to:
// ClassLoader loader = ExampleApplication.class.getClassLoader();
// URL imageURL = loader.getResource("com/example/icon.png");

Image image = ImageIO.read(imageURL);
}
}

```

リソースは、クラスがパッケージにかれるのとじて、.jarファイルのルートではなく、きパッケージにするがあります。たとえば、の.jarファイルがクラスパスにあり、の.jarファイルがルートにconfig.propertiesエントリをむ、getResourceメソッドまたはgetResourceAsStreamメソッドをびすと、にリストされている.jarからconfig.propertiesがされますクラスパスJava EEのように、クラスパスがアプリケーションののにないでは、これはなではありません。

すべてのgetResourceおよびgetResourceAsStreamメソッドは、されたリソースがしないはnullしnull。ビルドにアプリケーションにリソースをするがあるため、コードがきまれるときにリソースのをるがあります。にリソースをつけることができないのは、プログラマーのエラーのです。

リソースはみりです。リソースにきむはありません。のは、IDEEclipseなどでするときにリソースがのファイルであるため、なケースではのファイルのようにうことがであるとしていをするがよくあります。しかし、これはしくありません。アプリケーションはほとんどの、.jarまたは、warファイルなどのアーカイブとしてされます。そのような、リソースはのファイルではなく、きみではありません。URLクラスのgetFileメソッドはこれをするではなく、にもかかわらずURLのパスをすだけであり、これはしてなファイルであることはされません。

にリソースをリストするなはありません。ここでも、はビルドにリソースファイルをアプリケーションにするがあるため、はにそのパスをとっているはずですが、がなく、にはします。

Examples

リソースからイメージをロードする

バンドルされたイメージをロードするには

```

package com.example;

public class ExampleApplication {
    private Image getIcon() throws IOException {
        URL imageURL = ExampleApplication.class.getResource("icon.png");
        return ImageIO.read(imageURL);
    }
}

```

デフォルトのみみ

デフォルトのプロパティーをみるには

```
package com.example;

public class ExampleApplication {
    private Properties getDefaults() throws IOException {
        Properties defaults = new Properties();

        try (InputStream defaultsStream =
            ExampleApplication.class.getResourceAsStream("config.properties")) {

            defaults.load(defaultsStream);
        }

        return defaults;
    }
}
```

のJARからじのリソースをロードする

じパスとをつリソースは、クラスパスののJARファイルにするがあります。なケースは、にったリソースまたはパッケージのであるリソースです。そのようなリソースのはのとおりです。

- META-INF / MANIFEST.MF
- META-INF / beans.xmlCDI
- プロバイダをむServiceLoaderプロパティ

のでこれらのすべてのリソースへのアクセスをるためには、このためのメソッドをつクラスローダをするがあります。されたEnumerationは、CollectionsをしてListにできます。

```
Enumeration<URL> resEnum = MyClass.class.getClassLoader().getResources("META-INF/MANIFEST.MF");
ArrayList<URL> resources = Collections.list(resEnum);
```

クラスローダーをしたリソースのとみり

Javaでのリソースのロードは、のでされます。

1. リソースをつけるClassまたはClassLoader。
2. リソースをつける。
3. リソースのバイトストリームをします。
4. バイトストリームのみりと。
5. バイトストリームをします。

の3つのステップは、URLをライブラリメソッドまたはコンストラクタにしてリソースをロードすることによってされます。、このはgetResourceメソッドをしgetResource。アプリケーションコードのリソースデータをみることもできます。、このはgetResourceAsStreamをし

およびリソースパス

クラスパスからロードできるリソースは、パスでされます。パスのは、UNIX / Linuxファイルパスにしています。スラッシュ / でられたなでされています。パスはでまり、パスはセパレータでまります。

クラスパスののでしているように、JVMのクラスパスは、クラスパスにディレクトリとJARまたはZIPファイルのをオーバーレイすることによってをします。パスがされると、クラスローダーはの / をのルートをするものとしてをします。に、パスはの「フォルダ」にしてされるがあります。されるフォルダは、パスをするためにするオブジェクトによってなります。

クラスまたはクラスローダーの

リソースは、ClassオブジェクトまたはClassLoaderオブジェクトのいずれかをしてClassLoaderます。Classオブジェクトはパスをすることができるため、クラスリソースがあるは、はこれらのいずれかをしてをします。Classオブジェクトをするにはさまざまながあります。えば

- クラスリテラルは、JavaソースコードでをけることができるのクラスのClassオブジェクトをしてをします。えばString.classはStringのClassオブジェクトをしてをします。
- Object.getClass() は、のオブジェクトタイプのClassオブジェクトをしてをします。えば"hello".getClass() は、StringのClassをするのです。
- Class.forName(String) メソッドはにじてクラスをにロードし、そのClassオブジェクトをします。Class.forName("java.lang.String")。

ClassLoaderオブジェクトは、、ClassオブジェクトにしてgetClassLoader()をびすることによってされます。なClassLoader.getSystemClassLoader() メソッドをして、JVMのデフォルトクラスローダーをすることもできます。

get メソッド

ClassまたはClassLoaderインスタンスをClassLoaderたら、のいずれかのでリソースをつけることができます。

メソッド	
ClassLoader.getResource(path) ClassLoader.getResources(path)	されたパスをつリソースのをすURLをしてをします。
ClassLoader.getResources(path) Class.getResources(path)	foo.bar リソースのにできるURLをす Enumeration<URL>をしてをします。。

メソッド

`ClassLoader.getResourceAsStream(path)`
`Class.getResourceStream(path)`

`foo.bar` リソースのをのバイトとしてみる事ができ
る `InputStream` をします。

ノート

- `ClassLoader` と `Class` バージョンのメソッドのないは、パスがされるです。
 - `Class` メソッドは、クラスパッケージにする "フォルダ" のパスをします。
 - `ClassLoader` メソッドは、パスをパスとしています。つまり、クラスパスの "ルートフォルダ" でそれらをします。
- されたリソースまたはリソースが見つからない、 `getResource` および `getResourceAsStream` methods return `null` を methods return , and the `getResources` methods return an empty `Enumeration` を methods return an empty ます。
- されるURLは、 `URL.toStream()` をしてできます。それらは `file: URL` やのURLでもかまいませんが、リソースがJARファイルにある、JARファイルとそののリソースをする `jar: URL` になります。
- コードが `getResourceAsStream` メソッドまたは `URL.toStream()` をして `InputStream` をするは、ストリームオブジェクトをじるがあります。ストリームをしないと、リソースがリークするがあります。

オンラインでリソースクラスパスをむ <https://riptutorial.com/ja/java/topic/2433/> リソース-クラスパス-

128: リテラル

き

Javaリテラルは、をすつまり、Javaプログラムのソースコードにあるものです。としては、`1`、`0.333F`、`false`、`'X'`と`"Hello world\n"`。

Examples

16、8、および2リテラル

`hexadecimal`は`hexadecimal`のです。16の`0-9`と`AF`とはありませんがあります。 `AF`は`10-16`す。

`octal`は`octal`ので、`0-7`のがされます。

`binary`は、2ので、`0`と`1`がされます。

これらのはすべてじ`110`ます。

```
int dec = 110;           // no prefix --> decimal literal
int bin = 0b1101110;     // '0b' prefix --> binary literal
int oct = 0156;          // '0' prefix --> octal literal
int hex = 0x6E;          // '0x' prefix --> hexadecimal literal
```

バイナリリテラルがJava 7でされたことにしてください。

8のリテラルは、なエラーのトラップになりやすい。のリテラルにする、`0`、をすると、ったがされます。

```
int a = 0100;           // Instead of 100, a == 64
```

みやすくするためにアンダースコアをする

Java 7、プリミティブなリテラルののグループをるために、1つのアンダースコア_をしてをさせることができました。

たとえば、の2つのはです。

Java SE 7

```
int i1 = 123456;
int i2 = 123_456;
System.out.println(i1 == i2); // true
```

これは、にすように、すべてのプリミティブリテラルにできます。

Java SE 7

```
byte color = 1_2_3;
short yearsAnnoDomini= 2_016;
int socialSecurityNumber = 999_99_9999;
long creditCardNumber = 1234_5678_9012_3456L;
float piFourDecimals = 3.14_15F;
double piTenDecimals = 3.14_15_92_65_35;
```

これは、2、8、16のをしてもします。

Java SE 7

```
short binary= 0b0_1_0_1;
int octal = 07_7_7_7_7_7_7_0;
long hexBytes = 0xFF_EC_DE_5E;
```

のでのをするについてのいくつかのがあります

- のまたはわりにえは`_123`または`123_`ではありません
- リテラルのにしてえは`1._23`または`1_.23`ではありません
- FまたはLにちえは`1.23_F`または`99999999_L`ではありません
- のがされているでえは`0_xFFFF`ではありません

リテラルのエスケープシーケンス

とリテラルは、リテラルではされないなコードをにするエスケープメカニズムをします。エスケープシーケンスは、バックスラッシュ `\` と1つののでされます。リテラルのでじシーケンスがです。

エスケープシーケンスのなセットはのとおりです。

エスケープシーケンス	
<code>\\</code>	バックスラッシュ <code>\</code> をします。
<code>\'</code>	<code>'</code> をします
<code>\"</code>	<code>"</code> をします
<code>\n</code>	<code>LF</code> をします。
<code>\r</code>	キャリッジリターン <code>CR</code> をします。
<code>\t</code>	タブ <code>HT</code> をします
<code>\f</code>	フォームフィード <code>FF</code> をします。
<code>\b</code>	バックスペース <code>BS</code> をします。

\<octal>

0255ののコードをします。

の<octal>は025510のをす1,2または3つの8 '0'から '7'でされます。

バックスラッシュにけてのがくことは、なエスケープシーケンスであることにしてください。なエスケープシーケンスは、JLSによってコンパイルエラーとしてわれます。

- [JLS 3.10.6. およびリテラルのエスケープシーケンス](#)

Unicodeエスケープ

のとのエスケープシーケンスにえて、Javaには[JLS 3.3](#)でされているよりなUnicodeエスケープメカニズムがあります。 [Unicodeエスケープ](#)。 Unicodeエスケープのはのとおります。

```
'\ 'u' <hex-digit> <hex-digit> <hex-digit> <hex-digit>
```

ここで<hex-digit>のつである '0' '1' '2' '3' '4' '5' '6' '7' '8' '9' 'a' 'A' 'B' 'C' 'D' 'E' 'F' 'a' 、 'b' 'a' 、 'b' 、 'c' 、 'd' 、 'e' 、 'f' 、

Unicodeエスケープは、Javaコンパイラによってには16ビットのUnicode コード にマッピングされ、マッピングされたがなソースコードのどこでもできます。リテラルでASCIIをするがあるは、とりテラルでよくされます。

でエスケープする

10リテラル

リテラルは、 byte 、 short 、 int 、 longまたはcharインスタンスがなでできるをします。 このでは、なをにします。のでは、8、16、2のリテラルのいと、をめるためのアンダースコアのについてしています。

のリテラル

リテラルのもでなは、10リテラルです。えは

```
0 // The decimal number zero (type 'int')
1 // The decimal number one (type 'int')
42 // The decimal number forty two (type 'int')
```

ゼロにするがあります。のゼロは、リテラルを10ではなく8としてします。

```
077 // This literal actually means 7 x 8 + 7 ... or 63 decimal!
```

リテラルはなしです。あなたがかかれているは`-10`または`+10`、これらはにはしてされている`-`と`+`を。

このフォームのリテラルのは、`int`ののをち、`0`から`231`または`2,147,483,648`のでなければなりません。

`231`は`Integer.MAX_VALUE`より`1`きいことにしてください。 `0`リテラル`2147483647`どこでもすることができ、するためにコンパイルエラーである`2147483648`なし`-`。つまり、`Integer.MIN_VALUE`のをするためにされています。

```
int max = 2147483647;    // OK
int min = -2147483648;   // OK
int tooBig = 2147483648; // ERROR
```

いリテラル

`long`のリテラルは、`L`をしてします。例えば

```
0L          // The decimal number zero      (type 'long')
1L          // The decimal number one       (type 'long')
2147483648L // The value of Integer.MAX_VALUE + 1

long big = 2147483648;    // ERROR
long big2 = 2147483648L; // OK
```

`int`リテラルと`long`リテラルのは、のでもであることにしてください。例えば

```
int i = 2147483647;
long l = i + 1;           // Produces a negative value because the operation is
                          // performed using 32 bit arithmetic, and the
                          // addition overflows
long l2 = i + 1L;         // Produces the (intuitively) correct value.
```

JLS 3.10.1 - リテラル

ブールリテラル

ブールリテラルは、Javaプログラミングのリテラルのでもシンプルです。2つのな`boolean`は、`true`と`false`リテラルでされ`false`。とがされます。例えば

```
boolean flag = true;    // using the 'true' literal
flag = false;           // using the 'false' literal
```

リテラル

リテラルは、Javaソースコードののをすもなをします。リテラルはのものでされます。

- " でまります。
- またはではない0のの。 バックスラッシュ \ はののをします; [リテラルのエスケープシーケンス](#)をしてください。
- じた。

えば

```
"Hello world"    // A literal denoting an 11 character String
""               // A literal denoting an empty (zero length) String
 "\""           // A literal denoting a String consisting of one
                //      double quote character
"1\t2\t3\n"     // Another literal with escape sequences
```

1つのリテラルがのソースコードにまたがらないことにしてください。これは、リテラルののにまたはソースファイルのわりがするコンパイルエラーです。えば

```
"Jello world    // Compilation error (at the end of the line!)
```

い

にまらないがなは、それをのリテラルにし、 + をしてします。えば

```
String typingPractice = "The quick brown fox " +
                        "jumped over " +
                        "the lazy dog"
```

のようなリテラルと+されるは、 のをたします。これは、がコンパイラによってされ、にのStringオブジェクトによってされることをします。

リテラルのインターン

リテラルをむクラスファイルがJVMによってロードされると、するStringオブジェクトはランタイムシステムによってインメモリーされます。これは、のクラスでされるリテラルが、1つのクラスでされたよりもスペースをしないことをします。

インターンとストリングプールのについては、「」トピックの「[ストリングプールとヒープの](#)」をしてください。

Null リテラル

nullリテラル nullとしては、nullののをします。はです

```
MyClass object = null;
MyClass[] objects = new MyClass[]{new MyClass(), null, new MyClass()};

myMethod(null);
```

```
if (objects != null) {
    // Do something
}
```

`null`はむしろいいものです。はないので、Javaのソースコードですることはできません。また、のありません。

マルチタイプののは、のタイプになることです `null`。これは、すべてのとりてがあり、のにキャストできます。の、キャストはのチェックをとしません。

に、`null`は、がであっても `null instanceof <SomeReferenceType>` が `false` にされるというプロパティを持っています。

リテラル

リテラルは、`float` または `double` インスタンスがなにができるをします。リテラルには3あります。

- な10
- スケール
- 16

JLSでは、2つの10を1つのにしています。をにするため、々にいます。

をしてされた、`float` と `double` のなるリテラルがあります。々なでは、なるものをするためにをしています。これらのはをしません。

な10

リテラルのもなは、1つの10と、とオプションの `f`、`F`、`d` または `D` でされます。オプションのをすると、リテラルが `float f` または `F` または `double d` または `D` であることをできます。デフォルトをしないは `double` です。

例えば

```
0.0      // this denotes zero
.0       // this also denotes zero
0.       // this also denotes zero
3.14159  // this denotes Pi, accurate to (approximately!) 5 decimal places.
1.0F     // a `float` literal
1.0D     // a `double` literal.  (`double` is the default if no suffix is given)
```

には、がにく10もリテラルです。

```
1F       // means the same thing as 1.0F
```

10リテラルのは、10でされるのにもいIEEEです。このなは、もいへのめをしてIEEEバイナリにされます。のなセマンティクスは、`Double.valueOf(String)` と `Float.valueOf(String)` されています

が、のにいがあことに `Float.valueOf(String)`。

スケール

スケールされたのは、`E`または`e`によってされたをつシンプルに10でされ、きがきます。は、のにすように、に10のをけるためのいです。 `float`と`double`リテラルをするオプションのもあります。ここではいくつかのをします。

```
1.0E1    // this means 1.0 x 10^1 ... or 10.0 (double)
1E-1D    // this means 1.0 x 10^(-1) ... or 0.1 (double)
1.0e10f   // this means 1.0 x 10^(10) ... or 10000000000.0 (float)
```

リテラルのサイズは、 `float`または`double` によってされます。スケールのがきすぎるかさすぎると、コンパイルエラーです。

16

Java 6からは、リテラルを16ですることができます。 16には、シンプルなどのにたがありますが、のがなります。

1. すべての16リテラルはゼロ `0` でまり、に`x`または`X`まります。
2. のただしではありません には、 `a`から`f`までの16とそれにするもまれます。
3. はであり、 `e`または`E`わりに`p` または`P` というでされます。は、10のではなく2のであるスケールリングをします。

ここではいくつかのをします。

```
0x0.0p0f   // this is zero expressed in hexadecimal form (`float`)
0xff.0p19  // this is 255.0 x 2^19 (`double`)
```

アドバイス16はほとんどのJavaプログラマーにはみがないので、あまりしないことをおめします。

Java 7からは、リテラルの3つのすべてのストリングにアンダースコアをすることができます。これは「」のにもされます。 [をさせるためのアンダースコアの](#)をしてください。

なケース

リテラルが、したでするにはきすぎるかさすぎるをすは、コンパイルエラーです。つまり、`+` `INF`または`-INF`にオーバーフローした、またはアンダーフローが`0.0`にオーバーフローするです。しかし、リテラルがゼロのをすことはです。

リテラルは、`INF`や`NaN`などのIEEE 754のリテラルをしません。それらをソースコードでするがあるは、 `java.lang.Float` および `java.lang.Double` されたをすることをおめし `java.lang.Double`。
`Float.NaN`、 `Float.NEGATIVE_INFINITY`、 および `Float.NEGATIVE_INFINITY` など `Float.POSITIVE_INFINITY`

。

リテラル

リテラルは、Javaソースコードで`char`をするものをします。リテラルは、

- シングルクオート`'`。
- `"`の。これは、またはであってはいけませんが、バックスラッシュ`\`によってされたエスケープシーケンスであってもかまいません。 [リテラルのエスケープシーケンス](#)をしてください。
- `\"`じた`\"`。

例えば

```
char a = 'a';  
char doubleQuote = "\"";  
char singleQuote = '\\';
```

リテラルのはコンパイルエラーです。

```
char newline = '  
// Compilation error in previous line  
char newLine = '\\n'; // Correct
```

オンラインでリテラルをむ <https://riptutorial.com/ja/java/topic/8250/リテラル>

129: リフレクションAPI

き

リフレクションは、JVMでされているアプリケーションのものをべたりしたりするがあるプログラムでよくされます。 [Java Reflection API](#)は、コンパイルにをらなくても、にクラス、インタフェース、フィールド、メソッドをできるようにするでされます。また、しいオブジェクトをインスタンスし、リフレクションをしてメソッドをびすこともできます。

パフォーマンス

リフレクションはパフォーマンスをさせるがあります。リフレクションなしでタスクをできないにのみしてください。

Javaチュートリアル[Reflection API](#)から

リフレクションにはにされるがまれるため、のJavaマシンのはできません。その、は、よりもがくなり、のアプリケーションでにびされるコードのセクションではけるべきです。

Examples

き

Reflection APIをすると、にコードのクラスをチェックし、コードをにびすことができます。これはにですが、びしがかどうかをコンパイラがにできないため、です。

なは、えられたクラスのpublicコンストラクタとメソッドをすることです

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Method;

// This is a object representing the String class (not an instance of String!)
Class<String> clazz = String.class;

Constructor<?>[] constructors = clazz.getConstructors(); // returns all public constructors of String
Method[] methods = clazz.getMethods(); // returns all public methods from String and parents
```

このをして、オブジェクトをインスタンスし、なるメソッドをにびすことができます。

となタイプ

なタイプのは、

- `getGenericParameterTypes()` をしてメソッドのパラメータを `getGenericParameterTypes()` ます。
- `getGenericReturnType()` をして、メソッドのりのをします。
- `getGenericType` をして、パブリックフィールドを `getGenericType` ます。

のは、3つのケースすべてでタイプをするをしています。

```
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.Type;
import java.util.List;
import java.util.Map;

public class GenericTest {

    public static void main(final String[] args) throws Exception {
        final Method method = GenericTest.class.getMethod("testMethod", Map.class);
        final Field field = GenericTest.class.getField("testField");

        System.out.println("Method parameter:");
        final Type parameterType = method.getGenericParameterTypes()[0];
        displayGenericType(parameterType, "\t");

        System.out.println("Method return type:");
        final Type returnType = method.getGenericReturnType();
        displayGenericType(returnType, "\t");

        System.out.println("Field type:");
        final Type fieldType = field.getGenericType();
        displayGenericType(fieldType, "\t");

    }

    private static void displayGenericType(final Type type, final String prefix) {
        System.out.println(prefix + type.getTypeName());
        if (type instanceof ParameterizedType) {
            for (final Type subtype : ((ParameterizedType) type).getActualTypeArguments()) {
                displayGenericType(subtype, prefix + "\t");
            }
        }
    }

    public Map<String, Map<Integer, List<String>>> testField;

    public List<Number> testMethod(final Map<String, Double> arg) {
        return null;
    }

}
```

これにより、のようになられます。

```
Method parameter:
    java.util.Map<java.lang.String, java.lang.Double>
        java.lang.String
        java.lang.Double
Method return type:
```

```

    java.util.List<java.lang.Number>
        java.lang.Number
Field type:
    java.util.Map<java.lang.String, java.util.Map<java.lang.Integer,
java.util.List<java.lang.String>>>
        java.lang.String
        java.util.Map<java.lang.Integer, java.util.List<java.lang.String>>
            java.lang.Integer
            java.util.List<java.lang.String>
                java.lang.String

```

メソッドのびし

リフレクションをすると、にオブジェクトのメソッドをびすことができます。

これは、Stringオブジェクトのメソッドをびすをしています。

```

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

String s = "Hello World!";

// method without parameters
// invoke s.length()
Method method1 = String.class.getMethod("length");
int length = (int) method1.invoke(s); // variable length contains "12"

// method with parameters
// invoke s.substring(6)
Method method2 = String.class.getMethod("substring", int.class);
String substring = (String) method2.invoke(s, 6); // variable substring contains "World!"

```

フィールドのと

Reflection APIをすると、にフィールドのをまたはすることができます。たとえば、APIのように、OSなどのについてなフィールドをすることができます。finalのようなをして、なフィールドをできるようにすることもできます。

これをうには、のようなでClassgetFieldメソッドをするがあります

```

// Get the field in class SomeClass "NAME".
Field nameField = SomeClass.class.getDeclaredField("NAME");

// Get the field in class Field "modifiers". Note that it does not
// need to be static
Field modifiersField = Field.class.getDeclaredField("modifiers");

// Allow access from anyone even if it's declared private
modifiersField.setAccessible(true);

// Get the modifiers on the "NAME" field as an int.
int existingModifiersOnNameField = nameField.getModifiers();

// Bitwise AND NOT Modifier.FINAL (16) on the existing modifiers
// Readup here https://en.wikipedia.org/wiki/Bitwise\_operations\_in\_C

```

```
// if you're unsure what bitwise operations are.
int newModifiersOnNameField = existingModifiersOnNameField & ~Modifier.FINAL;

// Set the value of the modifiers field under an object for non-static fields
modifiersField.setInt(nameField, newModifiersOnNameField);

// Set it to be accessible. This overrides normal Java
// private/protected/package/etc access control checks.
nameField.setAccessible(true);

// Set the value of "NAME" here. Note the null argument.
// Pass null when modifying static fields, as there is no instance object
nameField.set(null, "Hacked by reflection...");

// Here I can directly access it. If needed, use reflection to get it. (Below)
System.out.println(SomeClass.NAME);
```

フィールドのははるかにです。 [Fieldget](#)およびそのをしてをできます。

```
// Get the field in class SomeClass "NAME".
Field nameField = SomeClass.class.getDeclaredField("NAME");

// Set accessible for private fields
nameField.setAccessible(true);

// Pass null as there is no instance, remember?
String name = (String) nameField.get(null);
```

これにしてください

[ClassgetDeclaredField](#)をするは、このクラスをしてクラスのフィールドをしします。

```
class HackMe extends Hacked {
    public String iAmDeclared;
}

class Hacked {
    public String someState;
}
```

ここで、 `HackMe#iAmDeclared` はされたフィールドです。しかし、 `HackMe#someState` はスーパークラス `Hacked` からされているため、されたフィールドではありません。

コールコンストラクタ

コンストラクタオブジェクトの

のように、 `Class` オブジェクトから `Constructor` クラスをできます。

```
Class myClass = ... // get a class object
Constructor[] constructors = myClass.getConstructors();
```

`constructors`には、クラスでされたパブリックコンストラクタの`Constructor`インスタンスが1つあります。

アクセスしたいコンストラクタのなパラメータがっているのは、のコンストラクタをフィルタすることができます。のは、されたクラスのパブリックコンストラクタをします。このコンストラクタは、`Integer`をパラメータとしてります。

```
Class myClass = ... // get a class object
Constructor constructor = myClass.getConstructor(new Class[]{Integer.class});
```

コンストラクタがされたコンストラクタにしない、`NoSuchMethodException`がスローされます。

コンストラクタオブジェクトをするしいインスタンス

```
Class myClass = MyObj.class // get a class object
Constructor constructor = myClass.getConstructor(Integer.class);
MyObj myObj = (MyObj) constructor.newInstance(Integer.valueOf(123));
```

の

このをとしてえる

```
enum Compass {
    NORTH(0),
    EAST(90),
    SOUTH(180),
    WEST(270);
    private int degree;
    Compass(int deg){
        degree = deg;
    }
    public int getDegree(){
        return degree;
    }
}
```

Javaではenumクラスはのクラスとていますが、enumにはいくつかのがあります。さらに、すべてのをするであるフィールドと、`values()`および`valueOf(String)`つ2つのメソッドがあり`values()`。

Reflectionをしてこのクラスのすべてのフィールドをすると、これをすることができます

```
for(Field f : Compass.class.getDeclaredFields())
    System.out.println(f.getName());
```

はのようになります。

ENUM \$ VALUES

そこで、のクラスとに、Reflectionでenumクラスをべることができました。しかし、Reflection APIは3つのメソッドをします。

チェック

```
Compass.class.isEnum();
```

をすクラスにしてtrueをします。

をする

```
Object[] values = Compass.class.getEnumConstants();
```

Compass.valuesのようなすべてのenumのをしますが、インスタンスはありません。

チェック

```
for(Field f : Compass.class.getDeclaredFields()){
    if(f.isEnumConstant())
        System.out.println(f.getName());
}
```

enumであるすべてのクラスフィールドをします。

そのされたでクラスをする

クラスのをむStringがえられた、ClassオブジェクトはClass.forNameをとってアクセスできます

```
Class clazz = null;
try {
    clazz = Class.forName("java.lang.Integer");
} catch (ClassNotFoundException ex) {
    throw new IllegalStateException(ex);
}
```

Java SE 1.2

クラスをするがある forName 2のパラメータ、するClassLoader 3のパラメータをすることができます。

```
ClassLoader classLoader = ...
boolean initialize = ...
Class clazz = null;
try {
    clazz = Class.forName("java.lang.Integer", initialize, classLoader);
} catch (ClassNotFoundException ex) {
```

```
        throw new IllegalStateException(ex);
    }
}
```

リフレクションをしてオーバーロードされたコンストラクタをびす

するパラメータをしてのコンストラクタをびす

```
import java.lang.reflect.*;

class NewInstanceWithReflection{
    public NewInstanceWithReflection(){
        System.out.println("Default constructor");
    }
    public NewInstanceWithReflection( String a){
        System.out.println("Constructor :String => "+a);
    }
    public static void main(String args[]) throws Exception {

        NewInstanceWithReflection object =
        (NewInstanceWithReflection)Class.forName("NewInstanceWithReflection").newInstance();
        Constructor constructor = NewInstanceWithReflection.class.getDeclaredConstructor( new
        Class[] {String.class});
        NewInstanceWithReflection object1 =
        (NewInstanceWithReflection)constructor.newInstance(new Object[]{"StackOverFlow"});

    }
}
```

```
Default constructor
Constructor :String => StackOverFlow
```

1. `Class.forName` をしてクラスのインスタンスをデフォルトのコンストラクタをびします。
2. `Class array` としてパラメータのをして、クラスの `getDeclaredConstructor` をびし
`getDeclaredConstructor`
3. コンストラクタをし `newInstance`、パラメータ `value` を `Object array` としてして `newInstance` をし
ます。

プライベートとをするための **Reflection API** の

リフレクションは、なでにされているにです。リフレクションをすると、プライベートにアクセスしてをできます。

はコードスニペットですが、これはおめできません。

```
import java.lang.reflect.*;

public class ReflectionDemo{
    public static void main(String args[]){
        try{
            Field[] fields = A.class.getDeclaredFields();
            A a = new A();
            for ( Field field:fields ) {
                if(field.getName().equalsIgnoreCase("name")){

```



```

        field.setAccessible(true);
        field.set(a, "StackOverFlow");
        System.out.println("A.name="+field.get(a));
    }
    if(field.getName().equalsIgnoreCase("age")){
        field.set(a, 20);
        System.out.println("A.age="+field.get(a));
    }
    if(field.getName().equalsIgnoreCase("rep")){
        field.setAccessible(true);
        field.set(a, "New Reputation");
        System.out.println("A.rep="+field.get(a));
    }
    if(field.getName().equalsIgnoreCase("count")){
        field.set(a, 25);
        System.out.println("A.count="+field.get(a));
    }
}
} catch (Exception err) {
    err.printStackTrace();
}
}

class A {
    private String name;
    public int age;
    public final String rep;
    public static int count=0;

    public A(){
        name = "Unset";
        age = 0;
        rep = "Reputation";
        count++;
    }
}

```

```

A.name=StackOverFlow
A.age=20
A.rep=New Reputation
A.count=25

```

のシナリオでは、`private`はされたクラスのゲッターメソッドとセッターメソッドなしにはアクセスできません。 `final`はにりてすることはできません。

`Reflection`はれるのようにのがとのをえるためにされるがある。

`field.setAccessible(true)`は、のをするためのキーです。

ネストされたクラスのコンストラクタをびす

ネストされたクラスのインスタンスをするは、 [クラスgetDeclaredConstructor](#)をして、むクラスのクラスオブジェクトをのパラメータとしてするがあります。

```

public class Enclosing{

```

```

public class Nested{
public Nested(String a){
    System.out.println("Constructor :String => "+a);
}
}
public static void main(String args[]) throws Exception {
    Class<?> clazzEnclosing = Class.forName("Enclosing");
    Class<?> clazzNested = Class.forName("Enclosing$Nested");
    Enclosing objEnclosing = (Enclosing)clazzEnclosing.newInstance();
    Constructor<?> constructor = clazzNested.getDeclaredConstructor(new
Class[] {Enclosing.class, String.class});
    Nested objInner = (Nested)constructor.newInstance(new Object[] {objEnclosing,
"StackOverFlow"});
}
}

```

ネストされたクラスがである、このむインスタンスはありません。

プロキシ

プロキシにはReflectionとはあまりがありませんが、APIのです。これは、にインターフェイスのをします。これはモックアップサービスをするときにちます。

プロキシは、すべてのメソッドびしをインターセプトし、びしをでできる、いわゆるびしハンドラをしてされたインターフェイスのインスタンスです。

```

public class DynamicProxyTest {

    public interface MyInterface1{
        public void someMethod1();
        public int someMethod2(String s);
    }

    public interface MyInterface2{
        public void anotherMethod();
    }

    public static void main(String args[]) throws Exception {
        // the dynamic proxy class
        Class<?> proxyClass = Proxy.getProxyClass(
            ClassLoader.getSystemClassLoader(),
            new Class[] {MyInterface1.class, MyInterface2.class});
        // the dynamic proxy class constructor
        Constructor<?> proxyConstructor =
            proxyClass.getConstructor(InvocationHandler.class);

        // the invocation handler
        InvocationHandler handler = new InvocationHandler(){
            // this method is invoked for every proxy method call
            // method is the invoked method, args holds the method parameters
            // it must return the method result
            @Override
            public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
            {
                String methodName = method.getName();

                if(methodName.equals("someMethod1")){
                    System.out.println("someMethod1 was invoked!");
                }
            }
        };
    }
}

```

```

        return null;
    }
    if(methodName.equals("someMethod2")){
        System.out.println("someMethod2 was invoked!");
        System.out.println("Parameter: " + args[0]);
        return 42;
    }
    if(methodName.equals("anotherMethod")){
        System.out.println("anotherMethod was invoked!");
        return null;
    }
    System.out.println("Unkown method!");
    return null;
}
};

// create the dynamic proxy instances
MyInterface1 i1 = (MyInterface1) proxyConstructor.newInstance(handler);
MyInterface2 i2 = (MyInterface2) proxyConstructor.newInstance(handler);

// and invoke some methods
i1.someMethod1();
i1.someMethod2("stackoverflow");
i2.anotherMethod();
}
}

```

このコードのはのとおりで。

```

someMethod1 was invoked!
someMethod2 was invoked!
Parameter: stackoverflow
anotherMethod was invoked!

```

Evil Javaがリフレクションでハックする

Reflection APIは、JDKのデフォルトライブラリでもプライベートフィールドとファイナルフィールドのをするためにできます。これは、わかるように、よくられているいくつかのクラスのをするためにできます。

なこと

ののは、Reflectionでできないのフィールドでまります。これがJava SecurityManagerです。[java.lang.System](#)でのようにされています。

```
private static volatile SecurityManager security = null;
```

しかし、このコードをするとSystemクラスにはリストされません

```
for(Field f : System.class.getDeclaredFields())
    System.out.println(f);
```

のザッツ `fieldFilterMap` で `sun.reflect.Reflection` にマップおよびセキュリティを `System.class` とリ

フレクシオンをつのアクセスにしてそれらをします。したがって、 `SecurityManager` アクティブすることはできませんでした。

クレイジーストリングス

Java Stringは、 `String` クラスのインスタンスとしてJVMによってされます。ただし、によっては、JVMは、じインスタンスをしてヒープをします。これは、リテラルや、 `String.intern()` びすことで `"String.intern()"` されたにしてもします。したがって、コードに `"hello"` がある、にじオブジェクトインスタンスになります。

はであるはずですが、 `"な"` をってをすることはです。のは、 `value` フィールドをきえることによってのをするをしてい `value` 。

```
public class CrazyStrings {
    static {
        try {
            Field f = String.class.getDeclaredField("value");
            f.setAccessible(true);
            f.set("hello", "you stink!".toCharArray());
        } catch (Exception e) {
        }
    }
    public static void main(String args[]) {
        System.out.println("hello");
    }
}
```

だから、このコードは `"あなたがをつ"`

1 = 42

じえをIntegerクラスでうこともできます

```
public class CrazyMath {
    static {
        try {
            Field value = Integer.class.getDeclaredField("value");
            value.setAccessible(true);
            value.setInt(Integer.valueOf(1), 42);
        } catch (Exception e) {
        }
    }
    public static void main(String args[]) {
        System.out.println(Integer.valueOf(1));
    }
}
```

すべてがです

そして、 [このstackoverflowの](#)によると、にいいことをするためにリフレクシオンをうことができます。

```
public class Evil {
    static {
        try {
            Field field = Boolean.class.getField("FALSE");
            field.setAccessible(true);
            Field modifiersField = Field.class.getDeclaredField("modifiers");
            modifiersField.setAccessible(true);
            modifiersField.setInt(field, field.getModifiers() & ~Modifier.FINAL);
            field.set(null, true);
        } catch (Exception e) {
        }
    }
    public static void main(String args[]){
        System.out.format("Everything is %s", false);
    }
}
```

たちがここでやっていることは、JVMをなでさせることになることにしてください。これはにです。

オンラインでリフレクションAPIをむ <https://riptutorial.com/ja/java/topic/629/> リフレクションapi

130: リモートメソッドびしRMI

RMIには、クライアント、サーバー、およびリモート・インタフェースの3つのコンポーネントがあります。リモート・インタフェースは、サーバーがしなければならないメソッドをすることによって、クライアント - サーバーをします。インタフェースは、メソッドをできるようにサーバからえるがあります。サーバがするメソッド「サービス」をするために、インタフェースをクライアントにえるようにするがあります。

リモートインタフェースをするオブジェクトはすべて、サーバーのをたすことをとしています。したがって、サーバがクライアントのメソッドもびすことができるクライアント - サーバは、にはサーバ - サーバです。これは、サーバーが「クライアント」をコールバックできるので、コールバックとされます。これをにいて、そのようにするサーバーにしてクライアントをすることはされます。

リモート・インタフェースは、[Remote](#)するのインタフェースです。サーバーとしてするオブジェクトは、のをします。

1. [Remote](#)をする[UnicastRemoteObject](#)をすることにより、またはにリモートインタフェースをします。
2. [UnicastRemoteObject](#)するには、[UnicastRemoteObject#exportObject](#)にすことでのエクスポートされ[UnicastRemoteObject#exportObject](#)。
3. のいずれかをして、レジストリにバインドさ[Registry](#)またはに[Naming](#)。これは、のをするためになのは、さらなるスタブがRMIをしてされるためです。

プロジェクトセットアップでは、クライアントプロジェクトとサーバープロジェクトはにですが、それぞれがビルドパスにプロジェクトをしています。プロジェクトには、リモートインタフェースがまれています。

Examples

クライアント - サーバあるJVMのメソッドをのJVMからびす

リモートインタフェース

```
package remote;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteServer extends Remote {

    int stringToInt(String string) throws RemoteException;
}
```

リモートインタフェースをしているサーバ

```

package server;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

import remote.RemoteServer;

public class Server implements RemoteServer {

    @Override
    public int stringToInt(String string) throws RemoteException {

        System.out.println("Server received: \"" + string + "\"");
        return Integer.parseInt(string);
    }

    public static void main(String[] args) {

        try {
            Registry reg = LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
            Server server = new Server();
            UnicastRemoteObject.exportObject(server, Registry.REGISTRY_PORT);
            reg.rebind("ServerName", server);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}

```

サーバーのメソッドを呼び出すクライアントリモート

```

package client;

import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

import remote.RemoteServer;

public class Client {

    static RemoteServer server;

    public static void main(String[] args) {

        try {
            Registry reg = LocateRegistry.getRegistry();
            server = (RemoteServer) reg.lookup("ServerName");
        } catch (RemoteException | NotBoundException e) {
            e.printStackTrace();
        }

        Client client = new Client();
        client.callServer();
    }

    void callServer() {

```

```

    try {
        int i = server.stringToInt("120");
        System.out.println("Client received: " + i);
    } catch (RemoteException e) {
        e.printStackTrace();
    }
}
}

```

したサーバー "120"

けたクライアント 120

コールバック "クライアント"のメソッドをびす

このでは、2つのクライアントがサーバーをしていにをしします。1つのクライアントは、2のクライアントにされるをサーバにする。2のクライアントは、そのをにして、サーバをして1のクライアントにりす。のクライアントもです。サーバは、クライアントからされたが10のときにをしします。サーバからクライアントへのりにされたは、プロセスをバックトラックします。

1. ログインサーバーはをレジストリにバインドします。
2. クライアントはログインサーバーをし、そのでloginメソッドをびしlogin。に
 - ログインサーバーはクライアントをしします。これには、コールバックメソッドをつクライアントのスタブがまれます。
 - ログインサーバーは、サーバスタブ「」または「セッション」をし、クライアントにしてしします。これには、logoutメソッドこのではされていないをむメソッドをむサーバーのスタブがまれます。
3. クライアントは、クライアントのとint passIntしてサーバーのpassIntをびします。
4. サーバーは、そのintクライアントのhalfをびします。これは、サーバーによってされるまで、のびしとコールバックをしします。

リモートインタフェース

ログインサーバー

```

package callbackRemote;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteLogin extends Remote {

    RemoteConnection login(String name, RemoteClient client) throws RemoteException;
}

```

サーバー

```

package callbackRemote;

```



```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteConnection extends Remote {

    void logout() throws RemoteException;

    String passInt(String name, int i) throws RemoteException;
}
```

クライアント

```
package callbackRemote;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteClient extends Remote {

    void half(int i) throws RemoteException;
}
```

ログインサーバー

```
package callbackServer;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.HashMap;
import java.util.Map;

import callbackRemote.RemoteClient;
import callbackRemote.RemoteConnection;
import callbackRemote.RemoteLogin;

public class LoginServer implements RemoteLogin {

    static Map<String, RemoteClient> clients = new HashMap<>();

    @Override
    public RemoteConnection login(String name, RemoteClient client) {

        Connection connection = new Connection(name, client);
        clients.put(name, client);
        System.out.println(name + " logged in");
        return connection;
    }

    public static void main(String[] args) {

        try {
            Registry reg = LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
            LoginServer server = new LoginServer();
            UnicastRemoteObject.exportObject(server, Registry.REGISTRY_PORT);
            reg.rebind("LoginServerName", server);
        } catch (RemoteException e) {

```

```

        e.printStackTrace();
    }
}
}

```

サーバー

```

package callbackServer;

import java.rmi.NoSuchObjectException;
import java.rmi.RemoteException;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.server.Unreferenced;

import callbackRemote.RemoteClient;
import callbackRemote.RemoteConnection;

public class Connection implements RemoteConnection, Unreferenced {

    RemoteClient client;
    String name;

    public Connection(String name, RemoteClient client) {

        this.client = client;
        this.name = name;
        try {
            UnicastRemoteObject.exportObject(this, Registry.REGISTRY_PORT);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void unreferenced() {

        try {
            UnicastRemoteObject.unexportObject(this, true);
        } catch (NoSuchObjectException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void logout() {

        try {
            UnicastRemoteObject.unexportObject(this, true);
        } catch (NoSuchObjectException e) {
            e.printStackTrace();
        }
    }

    @Override
    public String passInt(String recipient, int i) {

        System.out.println("Server received from " + name + ":" + i);
        if (i < 10)
            return String.valueOf(i);
    }
}

```

```

        RemoteClient client = LoginServer.clients.get(recipient);
        try {
            client.half(i);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
        return String.valueOf(i);
    }
}

```

クライアント

```

package callbackClient;

import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

import callbackRemote.RemoteClient;
import callbackRemote.RemoteConnection;
import callbackRemote.RemoteLogin;

public class Client implements RemoteClient {

    RemoteConnection connection;
    String name, target;

    Client(String name, String target) {

        this.name = name;
        this.target = target;
    }

    public static void main(String[] args) {

        Client client = new Client(args[0], args[1]);
        try {
            Registry reg = LocateRegistry.getRegistry();
            RemoteLogin login = (RemoteLogin) reg.lookup("LoginServerName");
            UnicastRemoteObject.exportObject(client, Integer.parseInt(args[2]));
            client.connection = login.login(client.name, client);
        } catch (RemoteException | NotBoundException e) {
            e.printStackTrace();
        }

        if ("Client1".equals(client.name)) {
            try {
                client.connection.passInt(client.target, 120);
            } catch (RemoteException e) {
                e.printStackTrace();
            }
        }
    }

    @Override
    public void half(int i) throws RemoteException {

        String result = connection.passInt(target, i / 2);
    }
}

```

```

        System.out.println(name + " received: \"" + result + "\"");
    }
}

```

をする

1. ログインサーバーをします。
2. Client2 Client1 1097してクライアントをします。
3. Client1 Client2 1098してクライアントをします。

3つのJVMがあるため、は3つのコンソールにされます。ここではになっています

```

Client2がログインしました
Client1がログインしました
クライアント1からしたサーバー120
クライアント2からしたサーバー60
クライアント1からしたサーバー30
クライアント2からしたサーバー15
Client1からしたサーバー7
したClient1「7」
したClient2 "15"
したClient1 "30"
したClient2 "60"

```

クライアントとサーバーのによるな**RMI**の

これは、5つのJavaクラスと2つのパッケージ サーバーとクライアントをつなRMIのです。

サーバーパッケージ

PersonListInterface.java

```

public interface PersonListInterface extends Remote
{
    /**
     * This interface is used by both client and server
     * @return List of Persons
     * @throws RemoteException
     */
    ArrayList<String> getPersonList() throws RemoteException;
}

```

PersonListImplementation.java

```

public class PersonListImplementation
extends UnicastRemoteObject
implements PersonListInterface
{

```

```

private static final long serialVersionUID = 1L;

// standard constructor needs to be available
public PersonListImplementation() throws RemoteException
{

/**
 * Implementation of "PersonListInterface"
 * @throws RemoteException
 */
@Override
public ArrayList<String> getPersonList() throws RemoteException
{
    ArrayList<String> personList = new ArrayList<String>();

    personList.add("Peter Pan");
    personList.add("Pippi Langstrumpf");
    // add your name here :)

    return personList;
}
}

```

Server.java

```

public class Server {

/**
 * Register servicer to the known public methods
 */
private static void createServer() {
    try {
        // Register registry with standard port 1099
        LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
        System.out.println("Server : Registry created.");

        // Register PersonList to registry
        Naming.rebind("PersonList", new PersonListImplementation());
        System.out.println("Server : PersonList registered");

    } catch (final IOException e) {
        e.printStackTrace();
    }
}

public static void main(final String[] args) {
    createServer();
}
}

```

クライアントパッケージ

PersonListLocal.java

```

public class PersonListLocal {
    private static PersonListLocal instance;

```

```

private PersonListInterface personList;

/**
 * Create a singleton instance
 */
private PersonListLocal() {
    try {
        // Lookup to the local running server with port 1099
        final Registry registry = LocateRegistry.getRegistry("localhost",
            Registry.REGISTRY_PORT);

        // Lookup to the registered "PersonList"
        personList = (PersonListInterface) registry.lookup("PersonList");
    } catch (final RemoteException e) {
        e.printStackTrace();
    } catch (final NotBoundException e) {
        e.printStackTrace();
    }
}

public static PersonListLocal getInstance() {
    if (instance == null) {
        instance = new PersonListLocal();
    }

    return instance;
}

/**
 * Returns the servers PersonList
 */
public ArrayList<String> getPersonList() {
    if (instance != null) {
        try {
            return personList.getPersonList();
        } catch (final RemoteException e) {
            e.printStackTrace();
        }
    }

    return new ArrayList<>();
}
}

```

PersonTest.java

```

public class PersonTest
{
    public static void main(String[] args)
    {
        // get (local) PersonList
        ArrayList<String> personList = PersonListLocal.getInstance().getPersonList();

        // print all persons
        for(String person : personList)
        {
            System.out.println(person);
        }
    }
}

```

アプリケーションのテスト

- `Server.java`の`main`メソッドをします。

```
Server : Registry created.  
Server : PersonList registered
```

- `PersonTest.java`の`main`メソッドをします。

```
Peter Pan  
Pippi Langstrumpf
```

オンラインでリモートメソッド呼び出しRMIをむ <https://riptutorial.com/ja/java/topic/171/リモートメソッド呼び出し-rmi->

131: リンクされたハッシュマップ

き

LinkedHashMapクラスは、なをして、Mapインタフェースのハッシュテーブルとリンクリストのです。HashMapクラスをし、Mapインターフェイスをします。

Java LinkedHashMapクラスにするなはのとおりです。LinkedHashMapには、キーにづくがまれています。のみがまれています。1つのヌルキーとのヌルをつことがあります。これは、わりにをするHashMapとじです。

Examples

Java LinkedHashMap クラス

キーポイント-

- ハッシュテーブルとリンクされたリストインプリメンテーションは、なでMapインターフェイスをします。
- HashMapクラスをし、Mapインターフェイスをします。
- キーにづくがまれています。
- ユニークなのみ。
- 1つのヌルキーとのヌルをつことができます。
- わりにHashMapとじようにをします。

-
- void clear。
- boolean containsKeyObject key
- Object getObject key
- protected boolean removeEldestEntryMap.Entry eldest
-

```
public static void main(String arg[])
{
    LinkedHashMap<String, String> lhm = new LinkedHashMap<String, String>();
    lhm.put("Ramesh", "Intermediate");
    lhm.put("Shiva", "B-Tech");
    lhm.put("Santosh", "B-Com");
    lhm.put("Asha", "Msc");
    lhm.put("Raghu", "M-Tech");
}
```



```
Set set = lhm.entrySet();
Iterator i = set.iterator();
while (i.hasNext()) {
    Map.Entry me = (Map.Entry) i.next();
    System.out.println(me.getKey() + " : " + me.getValue());
}

System.out.println("The Key Contains : " + lhm.containsKey("Shiva"));
System.out.println("The value to the corresponding to key : " + lhm.get("Asha"));
}
```

オンラインでリンクされたハッシュマップをむ <https://riptutorial.com/ja/java/topic/10750/>リンクされたハッシュマップ

132: ローカライゼーションと

Javaには、アプリケーションをローカライズするためのメカニズムがされていますが、ユーザーのロケールをしたりしたりするプログラムをしてしまうこともあります。

あなたのユーザーは、していたフォーマットにローカライズされたデータを見ることになり、これをサポートしようとするのはかなです。ここでは、「に」のされているとされるコンテンツをユーザーがすることをするさまざまなのさなをします。

ブラジル					
エジプト					
メキシコ	20/3/16	1.234,56	\$ 1,000.50	1,00050ドル	
	20/3/16	1,234.56	£1,000.50		100 km
	3/20/16	1,234.56	\$ 1,000.50	1,000.50 MXN	60 mi

リソース

- [Wikipedia とローカライゼーション](#)

Java リソース

- [Javaチュートリアル](#)
- [Oracle Javaプラットフォームにおけるロケールの](#)
- [JavaDoc Locale](#)

Examples

「ロケール」をしてにフォーマットされた

ピンチでSimpleDateFormatはらしいですが、のように、スケールがうまくいかないことがされています。

アプリケーションに"MM/dd/yyyy"をハードコードすると、ユーザーはせにならないでしょう。

Javaがあなたのためにをするようにしましょう

`DateFormat`のstaticメソッドをして、ユーザーのしいをします。デスクトップアプリケーション [デフォルトのロケール](#)にするのは、のようにびします。

```
String localizedDate = DateFormat.getDateInstance(style).format(date);
```

`style`は、`DateFormat`された `FULL`、`LONG`、`MEDIUM`、`SHORT`などの1 `DateFormat`です。

ユーザーがロケールをリクエストのとしてするサーバーアプリケーションの、わりに `getDateInstance()`ににすがあります。

```
String localizedDate =  
    DateFormat.getDateInstance(style, request.getLocale()).format(date);
```

の

をしない2つの

```
"School".equalsIgnoreCase("school"); // true
```

しないでください

```
text1.toLowerCase().equals(text2.toLowerCase());
```

はとをするためのなるをとっています。「」はで「」にされます。しかし、トルコでは「」は「ı」になります。`toLowerCase()`をするがあるは、`Locale String.toLowerCase(Locale)`をするオーバーロードをします。

いいをして2つのをする

```
Collator collator = Collator.getInstance(Locale.GERMAN);  
collator.setStrength(Collator.PRIMARY);  
collator.equals("Gärten", "gaerten"); // returns true
```

をして、のにつたをソートしますキーをして

```
String[] texts = new String[] {"Birne", "äther", "Apfel"};  
Collator collator = Collator.getInstance(Locale.GERMAN);  
collator.setStrength(Collator.SECONDARY); // ignore case  
Arrays.sort(texts, collator::compare); // will return {"Apfel", "äther", "Birne"}
```

ロケール

`java.util.Locale`クラスは、されたテキスト、、またはをローカライズするための「、または」をするためにされます。したがって、`Locale`オブジェクトは、、、さらにはのをむことができます。えは、あるののでされているか、そのがするとはなるでされます。

`Locale`インスタンスは、の、の、のがあるなど、アクションをローカライズするがあるコンポー

メントにされます。 `Locale`クラスは、それでやローカライズをうことはできません

は、ISO 639の2または3のコード、または8のみサブタグでなければなりません。に2と3のコードがあるは、2のコードをします。コードのなリストはIANA Language Subtag Registryにあります。

コードはとをしませんが、`Locale`クラスはにのコードをします

ロケールの

`java.util.Locale`インスタンスをするには、の4つのがあります。

```
Locale constants
Locale constructors
Locale.Builder class
Locale.forLanguageTag factory method
```

Java ResourceBundle

のような`ResourceBundle`インスタンスをします。

```
Locale locale = new Locale("en", "US");
ResourceBundle labels = ResourceBundle.getBundle("i18n.properties");
System.out.println(labels.getString("message"));
```

がプロパティファイル `i18n.properties` を持っているとしします

```
message=This is locale
```

```
This is locale
```

ロケールの

のをしてをしたいは、`setDefault()`メソッドをできます。その

```
setDefault(Locale.JAPANESE); //Set Japanese
```

オンラインでローカリゼーションとをむ <https://riptutorial.com/ja/java/topic/4086/ローカリゼーション>と

133: ローカルインナークラス

き

メソッドでされたクラスは、javaのローカルクラスとされます。ローカルのクラスのメソッドを
びすは、メソッドでこのクラスをインスタンスするがあります。

Examples

ローカルインナークラス

```
public class localInner1{
    private int data=30;//instance variable
    void display(){
        class Local{
            void msg(){System.out.println(data);}
        }
        Local l=new Local();
        l.msg();
    }
    public static void main(String args[]){
        localInner1 obj=new localInner1();
        obj.display();
    }
}
```

オンラインでローカルインナークラスをむ <https://riptutorial.com/ja/java/topic/10160/ローカルインナークラス>

134: ロギング `java.util.logging`

Examples

デフォルトログガーの

のは、デフォルトのlogging apiをするをしています。

```
import java.util.logging.Level;
import java.util.logging.Logger;

public class MyClass {

    // retrieve the logger for the current class
    private static final Logger LOG = Logger.getLogger(MyClass.class.getName());

    public void foo() {
        LOG.info("A log message");
        LOG.log(Level.INFO, "Another log message");

        LOG.fine("A fine message");

        // logging an exception
        try {
            // code might throw an exception
        } catch (SomeException ex) {
            // log a warning printing "Something went wrong"
            // together with the exception message and stacktrace
            LOG.log(Level.WARNING, "Something went wrong", ex);
        }

        String s = "Hello World!";

        // logging an object
        LOG.log(Level.FINER, "String s: {0}", s);

        // logging several objects
        LOG.log(Level.FINEST, "String s: {0} has length {1}", new Object[]{s, s.length()});
    }
}
```

ロギングレベル

JavaロギングApiには7 つのレベルがあります。のレベルはのとおりで。

- SEVERE
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST

デフォルトのレベルは `INFO` ただし、これはシステムによってなり、マシンをします。

レベル `OFF` ロギングをオフにするためにできると `ALL OFF` の `opposite` もあります。

このコード

```
import java.util.logging.Logger;

public class Levels {
    private static final Logger logger = Logger.getLogger(Levels.class.getName());

    public static void main(String[] args) {

        logger.severe("Message logged by SEVERE");
        logger.warning("Message logged by WARNING");
        logger.info("Message logged by INFO");
        logger.config("Message logged by CONFIG");
        logger.fine("Message logged by FINE");
        logger.finer("Message logged by FINER");
        logger.finest("Message logged by FINEST");

        // All of above methods are really just shortcut for
        // public void log(Level level, String msg):
        logger.log(Level.FINEST, "Message logged by FINEST");
    }
}
```

デフォルトでは、このクラスをすると、レベルが `CONFIG` よりいメッセージのみがされます。

```
Jul 23, 2016 9:16:11 PM LevelsExample main
SEVERE: Message logged by SEVERE
Jul 23, 2016 9:16:11 PM LevelsExample main
WARNING: Message logged by WARNING
Jul 23, 2016 9:16:11 PM LevelsExample main
INFO: Message logged by INFO
```

なメッセージのロギング

くのプログラムでられるログのサンプルをてみましょう。

```
public class LoggingComplex {

    private static final Logger logger =
        Logger.getLogger(LoggingComplex.class.getName());

    private int total = 50, orders = 20;
    private String username = "Bob";

    public void takeOrder() {
        // (...) making some stuff
        logger.fine(String.format("User %s ordered %d things (%d in total)",
                                   username, orders, total));
        // (...) some other stuff
    }

    // some other methods and calculations
}
```

```
}
```

のはにえますが、くのプログラマーはJava VMがスタックマシンであることをれています。つまり、メソッドをするに、すべてのメソッドのパラメータがされます。

このは、にFINE、FINER、FINESTようなデフォルトではになっているようなレベルのログをするために、Javaのログインにはです。takeOrder()メソッドのJavaバイトコードをてみましょう。

javap -c LoggingComplex.classのはのようになります。

```
public void takeOrder();
  Code:
    0: getstatic      #27 // Field logger:Ljava/util/logging/Logger;
    3: ldc           #45 // String User %s ordered %d things (%d in total)
    5: iconst_3
    6: anewarray      #3  // class java/lang/Object
    9: dup
   10: iconst_0
   11: aload_0
   12: getfield       #40 // Field username:Ljava/lang/String;
   15: astore
   16: dup
   17: iconst_1
   18: aload_0
   19: getfield       #36 // Field orders:I
   22: invokestatic   #47 // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
   25: astore
   26: dup
   27: iconst_2
   28: aload_0
   29: getfield       #34 // Field total:I
   32: invokestatic   #47 // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
   35: astore
   36: invokestatic   #53 // Method
java/lang/String.format:(Ljava/lang/String;[Ljava/lang/Object;)Ljava/lang/String;
   39: invokevirtual #59 // Method java/util/logging/Logger.fine:(Ljava/lang/String;)V
   42: return
```

39はのロギングをします。ロギングレベルがFINEよりもくデフォルトではされていれば、のすべてののロード、しいオブジェクトの、をformatメソッドではもできません。このようなロギングは、にであり、なメモリとプロセッサリソースをします。

そのため、するレベルがになっているかどうかをするがあります。

なはのとおりです。

```
public void takeOrder() {
    // making some stuff
    if (logger.isLoggable(Level.FINE)) {
        // no action taken when there's no need for it
        logger.fine(String.format("User %s ordered %d things (%d in total)",
                                   username, orders, total));
    }
    // some other stuff
}
```


Java 8

Loggerクラスには、`Supplier<String> as`パラメータをのメソッドがあります。これはラムダによってされます。

```
public void takeOrder() {  
    // making some stuff  
    logger.fine(() -> String.format("User %s ordered %d things (%d in total)",  
        username, orders, total));  
    // some other stuff  
}
```

Suppliers `get()` メソッド `get()` このはラムダは、するレベルがになったときにのみびされ、`if`はもうありません。

オンラインでロギング [java.util.logging](https://riptutorial.com/ja/java/topic/2010/ロギング-java-util-logging-) をむ <https://riptutorial.com/ja/java/topic/2010/ロギング-java-util-logging->

135: なJavaのとし

き

このトピックでは、Javaでののいのをします。

これには、Javaのないやランタイムのがまれます。

のAPIにするいは、それらのAPIにのトピックですることができます。はなケースです。それらはJavaでカバーされています。ないのは、このトピックの「」ですることができます。

Examples

とし==をしてなどのプリミティブラッパ―オブジェクトをする

このとしは、すべてのプリミティブラッパ―タイプにしくされますが、Integerおよびintについてはこれをします。

Integerオブジェクトをうときには、==をしてをすることがです。なぜなら、これはintをするためです。によってはこれがうまくいくようにえます

```
Integer int1_1 = Integer.valueOf("1");
Integer int1_2 = Integer.valueOf(1);

System.out.println("int1_1 == int1_2: " + (int1_1 == int1_2));           // true
System.out.println("int1_1 equals int1_2: " + int1_1.equals(int1_2));    // true
```

ここでは、1をつ2つのIntegerオブジェクトをしてしますこの、Stringとintリテラルの1つをします。また、2つの==とequalsがともtrueであることがわかりtrue。

このは、なるをするとします。

```
Integer int2_1 = Integer.valueOf("1000");
Integer int2_2 = Integer.valueOf(1000);

System.out.println("int2_1 == int2_2: " + (int2_1 == int2_2));           // false
System.out.println("int2_1 equals int2_2: " + int2_1.equals(int2_2));    // true
```

この、equalsのみがしいをもたらす。

こののいは、JVMがIntegerオブジェクトのキャッシュをしているためです。のは、システムプロパティ "java.lang.Integer.IntegerCache.high"でオーバーライドできます。JVM "-XX AutoBoxCacheMax = size"。こののの、Integer.valueOf()はしいをするのではなく、キャッシュされたをします。

したがって、のでは、Integer.valueOf(1)およびInteger.valueOf("1")びしはじキャッシュされた

`Integer` インスタンスをしました。に、2のでは、`Integer.valueOf(1000)` と `Integer.valueOf("1000")` ともし `Integer` オブジェクトをしてしました。

の`==`は、のつまり、じオブジェクトをします。したがって、のでは、がじで`true`ため `int1_1 == int1_2` は`true`です。2のでは、がなるため、`int2_1 == int2_2` は`false`です。

としリソースをれる

プログラムがファイルやネットワークなどのリソースをくたびに、リソースのがしたらリソースをすることがです。そのようなリソースのにがスローされる、のがです。`FileInputStream`は、ガベージコレクションイベントで`close()`メソッドをびす**ファイナライザ**があるとできます。ガベージコレクションのサイクルがいつされるかはわからないので、ストリームはコンピュータリソースをにすることがあります。try-catchブロックの`finally`セクションでリソースをじるがあります。

Java SE 7

```
private static void printFileJava6() throws IOException {
    FileInputStream input;
    try {
        input = new FileInputStream("file.txt");
        int data = input.read();
        while (data != -1){
            System.out.print((char) data);
            data = input.read();
        }
    } finally {
        if (input != null) {
            input.close();
        }
    }
}
```

Java 7、Java 7では、にこの、try-with-resourcesとばれるでなステートメントがされています。

Java SE 7

```
private static void printFileJava7() throws IOException {
    try (FileInputStream input = new FileInputStream("file.txt")) {
        int data = input.read();
        while (data != -1){
            System.out.print((char) data);
            data = input.read();
        }
    }
}
```

try-with-resourcesステートメントは、`Closeable`または`AutoCloseable`インターフェイスをするすべてのオブジェクトでできます。ステートメントのまでリソースがじられることをします。2つのインタフェースのいは、`Closeable close()`メソッドがらかのでされなければならない`IOException`をスローすることです。

リソースがにかれているが、ににじなければならないは、リソースをtry-with-resourcesのローカ

ルにりてることができます

Java SE 7

```
private static void printFileJava7(InputStream extResource) throws IOException {
    try (InputStream input = extResource) {
        ... //access resource
    }
}
```

try-with-resourcesコンストラクタでされたローカルリソースは、になものです。

としメモリリーク

Javaはメモリをにします。メモリをでするはありません。ヒープのオブジェクトのメモリは、オブジェクトがライブスレッドによってできなくなったときにガベージコレクタによってされるがあります。

ただし、になったオブジェクトにできるようにすることで、メモリのをぐことができます。これをメモリリークまたはメモリバックラッチングとんでも、はじです。りてられたメモリがにします。

Javaでのメモリリークはさまざまですが、もなは、ガベージコレクタはヒープからオブジェクトをできないため、なオブジェクトです。

フィールド

オブジェクトのコレクションをむstaticフィールドをつクラスをし、コレクションがになったにstaticフィールドをnullにするのをれることによって、そのようなをできます。staticフィールドはGCルートとみなされ、してされません。のは、JNIがされているときにヒープのメモリにリークがあることです。

クラスローダーのリーク

しかし、はるかになタイプのメモリリークは、[クラスローダのリーク](#)です。クラスローダーは、ロードしたすべてのクラスへのをし、すべてのクラスはそのクラスローダーへのをします。すべてのオブジェクトはそのクラスへのもしています。したがって、クラスローダによってロードされたクラスのオブジェクトでもガベージではない、そのクラスローダがロードしたのクラスをすることはできません。クラスはフィールドもするため、いずれもできません。

リークリークのは、のようになります。

```
final ScheduledExecutorService scheduledExecutorService = Executors.newScheduledThreadPool(1);
final Deque<BigDecimal> numbers = new LinkedBlockingDeque<>();
final BigDecimal divisor = new BigDecimal(51);

scheduledExecutorService.scheduleAtFixedRate(() -> {
    BigDecimal number = numbers.peekLast();
    if (number != null && number.remainder(divisor).byteValue() == 0) {
```

```

        System.out.println("Number: " + number);
        System.out.println("Deque size: " + numbers.size());
    }
}, 10, 10, TimeUnit.MILLISECONDS);

scheduledExecutorService.scheduleAtFixedRate(() -> {
    numbers.add(new BigDecimal(System.currentTimeMillis()));
}, 10, 10, TimeUnit.MILLISECONDS);

try {
    scheduledExecutorService.awaitTermination(1, TimeUnit.DAYS);
} catch (InterruptedException e) {
    e.printStackTrace();
}

```

これは、2つのスケジュールされたタスクをしますのタスクは、びされたキューからのを
`numbers`、そして、は51でりれるならば、それはやキューのサイズをします。2のタスクは、キューにをします。のタスクはレートでスケジュールされ、10ミリごとにされます。

コードがされると、キューのサイズがにすることがわかります。これにより、にキューがなすべてのヒープ・メモリーをするオブジェクトでいっぱいになります。

このプログラムのセマンティクスをしながらこれをするために、`deque pollLast`からをすることができます。 `peekLast` メソッドとは `peekLast`、 `pollLast` はをし、`deque`からしますが、 `peekLast` はののみをします。

とし`==`をってをする

Javaにとってよくあるいは、`==`をして2つのがしいかどうかをテストすることです。えば

```

public class Hello {
    public static void main(String[] args) {
        if (args.length > 0) {
            if (args[0] == "hello") {
                System.out.println("Hello back to you");
            } else {
                System.out.println("Are you feeling grumpy today?");
            }
        }
    }
}

```

のプログラムは、のコマンドラインをテストし、それが "hello" というではないときになるメッセージをすることになっています。しかしは、それがうまくいかないことです。そのプログラムは "あなたははしいがしますか" のコマンドラインがであっても。

このケースでは、 `String String args [0]` がヒープにあるに `String String "hello"` がプールにられます。これは、じリテラルをす2つのオブジェクトがあり、それぞれがをつことをします。 `==`、のではなく、をテストするので、はほとんどの、をします。これはにそうすることをするものではありません。

==をしてをテストする、にテストするのは、2つのStringオブジェクトがJavaオブジェクトであるです。ながら、これはJavaではのがするものではありません。、をテストするしいは equals(Object) メソッドをうこと equals(Object) 。 のペアについては、じでじでされているかどうかをテストします。

```
public class Hello2 {
    public static void main(String[] args) {
        if (args.length > 0) {
            if (args[0].equals("hello")) {
                System.out.println("Hello back to you");
            } else {
                System.out.println("Are you feeling grumpy today?");
            }
        }
    }
}
```

しかし、にはしています。は、いくつかので== がされるえをすということです。えば

```
public class Test1 {
    public static void main(String[] args) {
        String s1 = "hello";
        String s2 = "hello";
        if (s1 == s2) {
            System.out.println("same");
        } else {
            System.out.println("different");
        }
    }
}
```

いことに、をったでテストしていても、これは"じ"とされます。ですか [Javaセクション3.10.5リテラルでは](#)、じからなる2つの>>リテラル<<にはじJavaオブジェクトによってされるとされているためです。したがって、== testはしいリテラルにしてをえます。リテラルは"interned"され、コードがロードされるときに"プール"にされますが、はのです。

このにえて、Javaでは、2つのリテラルをするコンパイルのがある、それは1つのリテラルにするとされています。って

```
public class Test1 {
    public static void main(String[] args) {
        String s1 = "hello";
        String s2 = "hel" + "lo";
        String s3 = " mum";
        if (s1 == s2) {
            System.out.println("1. same");
        } else {
            System.out.println("1. different");
        }
        if (s1 + s3 == "hello mum") {
            System.out.println("2. same");
        } else {
            System.out.println("2. different");
        }
    }
}
```

```
}  
}
```

これは、"1. same"と "2. different"をします。のケースでは、コンパイルに+がされ、1つのStringオブジェクトとそれをします。2のケースでは、にされ、2つのなるStringオブジェクト

すると、Javaでをテストするために==をすることはほとんどにしていますが、ったえをえることはされません。

としファイルをこうとするにテストします。

ファイルをくに、さまざまなテストをファイルにして、よりいをしたり、をしたりすることをおめします。たとえば、このメソッドは、pathがみりなファイルにするかどうかをチェックしようとしします。

```
public static File getValidatedFile(String path) throws IOException {  
    File f = new File(path);  
    if (!f.exists()) throw new IOException("Error: not found: " + path);  
    if (!f.isFile()) throw new IOException("Error: Is a directory: " + path);  
    if (!f.canRead()) throw new IOException("Error: cannot read file: " + path);  
    return f;  
}
```

のをのようになうことができます

```
File f = null;  
try {  
    f = getValidatedFile("somefile");  
} catch (IOException ex) {  
    System.err.println(ex.getMessage());  
    return;  
}  
try (InputStream is = new FileInputStream(file)) {  
    // Read data etc.  
}
```

のは、FileInputStream(File)シグネチャにありFileInputStream(File)なぜなら、コンパイラはここでIOExceptionをまえようとしているか、スタックをさらにIOExceptionうとしているからです。

2のは、getValidatedFileによってされるチェックが、FileInputStreamがすることをしないことです。

- のスレッドまたはのプロセスが、getValidatedFileがしたでファイルのをしたり、ファイルをしたり、みりアクセスをすることができます。これは、カスタムメッセージのない「の」IOExceptionつながります。
- これらのテストでカバーされていないエッジケースがあります。たとえば、SELinuxがモードになっているシステムでは、canRead() trueしても、ファイルをみもうとしてもするがありtrue。

3のは、テストがであることです。たとえば、`exists`、`isFile`、および`canRead`は、それぞれがなチェックをするためにシステムコールをします。のシステムコールがファイルをくようになれ、じチェックがバックグラウンドでりされます。

するに、`getValidatedFile`のような`getValidatedFile`がっています。にファイルをいてをしようとするといでしょう

```
try (InputStream is = new FileInputStream("somefile")) {
    // Read data etc.
} catch (IOException ex) {
    System.err.println("IO Error processing 'somefile': " + ex.getMessage());
    return;
}
```

いてみんだときにスローされたIOエラーをしたいは、ネストされたtry / catchをできます。いてるにしてよりいをしたいは、ハンドラの`exists`、`isFile`、および`canRead`チェックをできます。

としオブジェクトとしてののえ

Javaはオブジェクトをしません。

```
String foo;    // NOT AN OBJECT
```

いずれのJavaにもオブジェクトはまれません。

```
String bar[] = new String[100]; // No member is an object.
```

ってをオブジェクトとえと、Javaののはあなたをさせるでしょう。

- プリミティブ `int` や `float` をつJavaの、にはのコピーがされます。プリミティブのすべてのコピーはできません。つまり、1の`int`は1つだけです。プリミティブはオブジェクトではなく、オブジェクトのようにしません。
- クラスまたはのいずれかをつJavaの、にはがされます。のすべてのコピーはできません。はオブジェクトをすかもしれないし、オブジェクトが`null`れていないことをする`null`かもしれない。しかし、それらはオブジェクトではなく、オブジェクトのようにしません。

いずれのでもはオブジェクトではなく、いずれのもオブジェクトはまれません。それらはオブジェクトへののをむかもしれませんが、それはかうものをつっています。

サンプルクラス

のでは、このクラスをしています。このクラスは、2Dのポイントをします。

```
public final class MutableLocation {
    public int x;
    public int y;
```



```

public MutableLocation(int x, int y) {
    this.x = x;
    this.y = y;
}

public boolean equals(Object other) {
    if (!(other instanceof MutableLocation)) {
        return false;
    }
    MutableLocation that = (MutableLocation) other;
    return this.x == that.x && this.y == that.y;
}
}

```

このクラスのインスタンスは、`int`の2つのフィールド`x`と`y`をつオブジェクトです。

たちは、`MutableLocation`クラスのくのインスタンスをつことができます。いくつかは2Dでじをします。すなわち、`x`と`y`それぞれのがする。はなるをします。

のかじオブジェクトをすことができる

```

MutableLocation here = new MutableLocation(1, 2);
MutableLocation there = here;
MutableLocation elsewhere = new MutableLocation(1, 2);

```

では、つのしている`here`、`there`と`elsewhere`へのをすることができます。`MutableLocation`オブジェクトを。

あなたがってこれらのをオブジェクトであるとえるならば、あなたはそのステートメントをのうにってむがあります

1. 「[1, 2]」を`here`コピーして`here`
2. 「[1, 2]」を`there`コピーし`there`
3. 「[1, 2]」を`elsewhere`のにコピーする

これから、3つのに3つのしたオブジェクトがあるとできます。には、でされたオブジェクトは2つだけです。`here`と`there`、にじオブジェクトをしてください。

これをすることができます。のをのうにします

```

System.out.println("BEFORE: here.x is " + here.x + ", there.x is " + there.x +
    "elsewhere.x is " + elsewhere.x);
here.x = 42;
System.out.println("AFTER: here.x is " + here.x + ", there.x is " + there.x +
    "elsewhere.x is " + elsewhere.x);

```

これはのうにされます

```

BEFORE: here.x is 1, there.x is 1, elsewhere.x is 1

```

```
AFTER: here.x is 42, there.x is 42, elsewhere.x is 1
```

たちは`here.x`しいをりて、そこからるをし`there.x`。らはじをしています。しかし、`elsewhere.x`される`elsewhere.x`はさ`elsewhere.x`ていないため、`elsewhere`のオブジェクトをするがあります。

がオブジェクトの、このりて`here.x = 42`は`there.x`されません。

は、2つのオブジェクトがしいことをテストしません

`==` をにすると、がじオブジェクトをしているかどうかテストされます。2つのなるオブジェクトがなで「しい」かどうかはテストされません。

```
MutableLocation here = new MutableLocation(1, 2);
MutableLocation there = here;
MutableLocation elsewhere = new MutableLocation(1, 2);

if (here == there) {
    System.out.println("here is there");
}
if (here == elsewhere) {
    System.out.println("here is elsewhere");
}
```

これは "here is there"とされますが、"here is other is"はされません。 `here`と`elsewhere`のは、2つのなるオブジェクトです。

に、でした`equals(Object)` メソッドをびすと、2つの`MutableLocation`インスタンスのがじかどうかをテストします。

```
if (here.equals(there)) {
    System.out.println("here equals there");
}
if (here.equals(elsewhere)) {
    System.out.println("here equals elsewhere");
}
```

のメッセージがされます。に、`here.equals(elsewhere)`は、2つの`MutableLocation`オブジェクトののためにしたがつたされているため、`true`し`true`。

メソッドびしはオブジェクトをまったくさない

Javaメソッドのびしは、をすと、をすために、`1`でパスをします。

メソッドにをすと、にはオブジェクトへのをでしています。つまり、オブジェクトのコピーをしています。

のオブジェクトがじオブジェクトをしているり、そのオブジェクトをいずれかのからすることができます。これがのオブジェクトののとなります。

しかし、によってオブジェクトをすることはできません²。は、オブジェクトコピーがのオブジェクトをすようにされた、のオブジェクトがとしてのオブジェクトをしすことである。

```
void f(MutableLocation foo) {
    foo = new MutableLocation(3, 4);    // Point local foo at a different object.
}

void g() {
    MutableLocation foo = MutableLocation(1, 2);
    f(foo);
    System.out.println("foo.x is " + foo.x); // Prints "foo.x is 1".
}
```

オブジェクトのコピーをすこともありません。

```
void f(MutableLocation foo) {
    foo.x = 42;
}

void g() {
    MutableLocation foo = new MutableLocation(0, 0);
    f(foo);
    System.out.println("foo.x is " + foo.x); // Prints "foo.x is 42"
}
```

1 - PythonやRubyのようなでは、オブジェクト/の"し"には"です"というがされます。

2 - 「し」または「によるびし」というは、プログラミングのではになをちます。つまり、びされたメソッドがにしいをするときに、ののをするように、またはのアドレスをすことをします。Javaはこれをサポートしていません。パラメータをすためのさまざまなメカニズムについては、https://en.wikipedia.org/wiki/Evaluation_strategyをしてください。

としりてとをみわせる

にはStackOverflowのJavaのおよびCまたはC++ので、のようなをすることがあります。

```
i += a[i++] + b[i--];
```

i、a、bのについていくつかします。

にえは

- Javaの、えはに¹とされていますが、ではなく、しばしばがしい
- CとC++の、えはしばしばです。

このようなは、やがJavaプログラミングでがにどのようにするかをするためのみとして、やでよくされます。これはおそらく「のテスト」としてなものですが、のプログラムでこれをうべきではありません。

するために、のながStackOverflowのこのようににしました。によっては、かのコードにのいと

してされます。

```
int a = 1;
a = a++;
System.out.println(a);    // What does this print.
```

これらのステートメントをすばやくんでいるプログラマーJavaをむのほとんどは、₂するといま
す。には、₁します。のについては、[この](#)をおみください。

しかし、これとのから、のおちりは、のJavaはともにりてるとがじがのでハードをすること、
そしてのにをくことになるだろうということです。このようなコードをくのはけるべきです。

1 - またはオブジェクトがのスレッドからえるは、[Javaメモリモデル](#)のなをモジュールにします。

とし**String**がクラスであるとしていない

しいJavaプログラマは、Java `String` クラスがであることをしばしばれるか、にしない。これは、
ののようにつながります。

```
public class Shout {
    public static void main(String[] args) {
        for (String s : args) {
            s.toUpperCase();
            System.out.print(s);
            System.out.print(" ");
        }
        System.out.println();
    }
}
```

のコードはコマンドラインのをですることになっています。ながら、それはしません、のはされ
ません。はのとおりです。

```
s.toUpperCase();
```

`toUpperCase()` をびすと、`s` がのにされるとえるかもしれません。それはしません。それはできません
ん `String` オブジェクトはです。らはすることはできません。

には、`toUpperCase()` メソッドは `String` オブジェクトをします `toUpperCase()` オブジェクトは、それ
をびす `String` バージョンです。これはおそろしい `String` オブジェクトになりますが、`s` がすでに
すべてであった、はのになります。

したがって、このメソッドをにするには、メソッドびしによってされたオブジェクトをするがあ
ります。えば

```
s = s.toUpperCase();
```

、「はされません」ルールはすべての `String` メソッドにされます。あなたがそれをえているなら

ば、のいをけることができます。

オンラインでなJavaのとしをむ <https://riptutorial.com/ja/java/topic/4388>/なjavaのとし

136: オブジェクト

オブジェクトはセッターなしなので、オブジェクトにはすべてのをるがあります。

にはではありませんが、すべてのフィールドを`final`にすることがベストプラクティスです。これにより、クラスがスレッドセーフになりますJava Concurrency in Practice、3.4.1を。

このは、これをするのについくつかのパターンをしています。

Examples

なコピーをしてタイプのバージョンをする。

Javaのなとクラスのにはになものがあります。たとえば、すべてのはであり、`java.util.Data`などのクラスもです。これは、がけられているではにくいことがあります。

これにする1つのは、のラッパーをすることです。ののなラッパーです

```
public class ImmutableIntArray {
    private final int[] array;

    public ImmutableIntArray(int[] array) {
        this.array = array.clone();
    }

    public int[] getValue() {
        return this.clone();
    }
}
```

このクラスはなコピーをして、な `int[]` をするのあるコードからします。

- コンストラクタは`clone()`をしてパラメータののコピーをします。でコンストラクタのびしがパラメータをした、それは`ImmutableIntArray`にしません。
- また、`getValue()` メソッドは`clone()`をしてされるをします。びしがをしたは、`ImmutableIntArray`にしません。

ラップされたにしてみりをするために、メソッドを`ImmutableIntArray`にすることもできます。え、さをしたり、のインデックスでをしたりするなどです。

このようにされたラッパーは、のとがないことにしてください。のためにをにきえることはできません。

クラスのレシピ

オブジェクトは、をできないオブジェクトです。クラスは、そのインスタンスがおよびによって

であるクラスです。のとしてもにされるJavaクラスは[java.lang.String](#)です。

は、ステレオタイプのです。

```
public final class Person {
    private final String name;
    private final String ssn;        // (SSN == social security number)

    public Person(String name, String ssn) {
        this.name = name;
        this.ssn = ssn;
    }

    public String getName() {
        return name;
    }

    public String getSSN() {
        return ssn;
    }
}
```

このバリエーションは、コンストラクタを`private`としてし、わりに`public static factory`メソッドをすることです。

クラスのレシピはのとおりです。

- すべてのプロパティは、コンストラクタまたはファクトリメソッドですがあります。
- セッターはいないはずで。
- インタフェースのために`setter`をめるがあるは、もしないか、をスローするがあります。
- すべてのプロパティは`private`で`final`としてするがあります。
- なへのであるすべてのプロパティ
 - プロパティは、コンストラクタをしてされたのディープコピーでされるがあります。
 - プロパティの`getter`はプロパティのコピーをすがあります。
- クラスは、クラスのなサブクラスをしないように`final`としてするがあります。

すべきのいくつかの

- は、オブジェクトがヌルであることをげません。えは`null`を`String`にりてることができます。
- クラスのプロパティが`final`としてされている、インスタスにはスレッドセーフです。これにより、クラスはマルチスレッドアプリケーションをするためのれたビルディングブロックになります。

クラスがにならないようにするなの

いくつかのセッターをして、コンストラクターになすべてのプロパティをせずに、

```
public final class Person { // example of a bad immutability
```

```

private final String name;
private final String surname;
public Person(String name) {
    this.name = name;
}
public String getName() { return name;}
public String getSurname() { return surname;}
public void setSurname(String surname) { this.surname = surname;}
}

```

Personクラスがでないことをすのはです

```

Person person = new Person("Joe");
person.setSurname("Average"); // NOT OK, change surname field after creation

```

これをするには、にsetSurname()をし、のようにコンストラクタをリファクタリングします。

```

public Person(String name, String surname) {
    this.name = name;
    this.surname = surname;
}

```

インスタンスをおよびとしてマークしない

のクラスをてください

```

public final class Person {
    public String name;
    public Person(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
}

```

のスニペットは、のクラスがでないことをしています

```

Person person = new Person("Average Joe");
person.name = "Magic Mike"; // not OK, new name for person after creation

```

これをするには、にnameプロパティをprivateとfinalとしてマークします。

クラスのオブジェクトをゲッターにする

のクラスをてください

```

import java.util.List;
import java.util.ArrayList;
public final class Names {

```



```

private final List<String> names;
public Names(List<String> names) {
    this.names = new ArrayList<String>(names);
}
public List<String> getNames() {
    return names;
}
public int size() {
    return names.size();
}
}

```

Names クラスは、`names` のようなもののコードがすようにではありません。

```

List<String> namesList = new ArrayList<String>();
namesList.add("Average Joe");
Names names = new Names(namesList);
System.out.println(names.size()); // 1, only containing "Average Joe"
namesList = names.getNames();
namesList.add("Magic Mike");
System.out.println(names.size()); // 2, NOT OK, now names also contains "Magic Mike"

```

これは、`getNames()` によってされたリストをすると、`Names` リストをできるためです。

これをするには、`namesList` をコピーをするか、`namesList` のようにクラスのオブジェクトをするをさないでください。

```

public List<String> getNames() {
    return new ArrayList<String>(this.names); // copies elements
}

```

または、`namesList` のようにのオブジェクトとプリミティブが、`namesList` をするでゲッターをすることによって

```

public String getName(int index) {
    return names.get(index);
}
public int size() {
    return names.size();
}

```

クラスでなオブジェクトをコンストラクタにする

これはのバリエーションです。のクラスをてください

```

import java.util.List;
public final class NewNames {
    private final List<String> names;
    public Names(List<String> names) {
        this.names = names;
    }
    public String getName(int index) {
        return names.get(index);
    }
}

```

```

    public int size() {
        return names.size();
    }
}

```

の `Names` クラスとに、`NewNames` クラスものであるようですが、にはのスニペットはそれとはのことをしています

```

List<String> namesList = new ArrayList<String>();
namesList.add("Average Joe");
NewNames names = new NewNames(namesList);
System.out.println(names.size()); // 1, only containing "Average Joe"
namesList.add("Magic Mike");
System.out.println(names.size()); // 2, NOT OK, now names also contains "Magic Mike"

```

これをするには、のと同じように、オブジェクトをクラスにりてずにコピーをするだけです。つまり、コンストラクタはのようになります。

```

public Names(List<String> names) {
    this.names = new ArrayList<String>(names);
}

```

クラスのメソッドをオーバーライドさせる

のクラスをてください

```

public class Person {
    private final String name;
    public Person(String name) {
        this.name = name;
    }
    public String getName() { return name;}
}

```

`Person` クラスは、のではのようですが、`Person` しいサブクラスがされているとします。

```

public class MutablePerson extends Person {
    private String newName;
    public MutablePerson(String name) {
        super(name);
    }
    @Override
    public String getName() {
        return newName;
    }
    public void setName(String name) {
        newName = name;
    }
}

```

`Person` `im`のは、しいサブクラスをしてをすることができます

```
Person person = new MutablePerson("Average Joe");
System.out.println(person.getName()); prints Average Joe
person.setName("Magic Mike"); // NOT OK, person has now a new name!
System.out.println(person.getName()); // prints Magic Mike
```

このをする、のいずれかのようにクラスマークするには `final` それをすることはできませんかのよう
にそのコンストラクタSのすべての `private`。

オンラインでオブジェクトをむ <https://riptutorial.com/ja/java/topic/2807/オブジェクト>

137: クラス

き

オブジェクトは、インスタンスがされたにがしないインスタンスです。たとえば、Stringはクラスであり、インスタンスされるとそのはしてされません。

Javaのいくつかのクラス

1. java.lang.String
2. プリミティブのラッパークラスjava.lang.Integer、java.lang.Byte、java.lang.Character、java.lang.Short、java.lang.Boolean、java.lang.Long、java.lang.Double、java.lang.Float
3. ほとんどのenumクラスはですが、これはなににします。
4. java.math.BigIntegerとjava.math.BigDecimalなくともクラスのオブジェクト
5. java.io.File。これは、VMのオブジェクトローカルシステムのファイルをし、するとしながあり、このオブジェクトのをおよびするいくつかのメソッドがあります。しかし、Fileオブジェクトはのままです。

Examples

クラスをするための

のは、オブジェクトをするためのなをしています。

1. フィールドでされるフィールドやオブジェクトをするメソッドである "setter"メソッドをしなないでください。
2. すべてのフィールドをかつプライベートにする。
3. サブクラスによるメソッドのオーバーライドをしなないでください。これをうもなは、クラスをfinalとしてすることです。よりされたアプローチは、コンストラクタをプライベートにして、ファクトリメソッドでインスタンスをすることです。
4. インスタンスフィールドになオブジェクトへののがまれているは、それらのオブジェクトのをしなないでください。
5. なオブジェクトをするメソッドをしなないでください。
6. なオブジェクトへのをしなないでください。コンストラクタにされたのなオブジェクトへのをしなないでください。にじてコピーをし、コピーへのをしします。に、メソッドでオリジナルをさないように、にじてオブジェクトのコピーをしします。

がない

```
public final class Color {  
    final private int red;  
    final private int green;  
    final private int blue;
```

```

private void check(int red, int green, int blue) {
    if (red < 0 || red > 255 || green < 0 || green > 255 || blue < 0 || blue > 255) {
        throw new IllegalArgumentException();
    }
}

public Color(int red, int green, int blue) {
    check(red, green, blue);
    this.red = red;
    this.green = green;
    this.blue = blue;
}

public Color invert() {
    return new Color(255 - red, 255 - green, 255 - blue);
}
}

```

の

この、クラス `Point` はであり、このクラスのオブジェクトのを作ることができるユーザもいます。

```

class Point {
    private int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getY() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
    }
}

//...

public final class ImmutableCircle {
    private final Point center;
    private final double radius;

    public ImmutableCircle(Point center, double radius) {
        // we create new object here because it shouldn't be changed
        this.center = new Point(center.getX(), center.getY());
        this.radius = radius;
    }
}

```

のはですか

のはにあります。のスレッドがじオブジェクトのをしようとしているがあり、じオブジェクトのなるをているスレッドにつながるがあるため、なオブジェクトのをすることはしいオブジェクト。

オブジェクトをつことによって、オブジェクトのがわからないので、オブジェクトをているすべてのスレッドがじをることをすることができます。

オンラインでクラスをむ <https://riptutorial.com/ja/java/topic/10561/> クラス

138: プログラミングスレッド

き

コンカレントコンピューティングは、いくつかのがではなくにされるコンピューティングのです。Javaは、スレッドのによるプログラミングをサポートするようにされています。オブジェクトとリソースにはのスレッドからアクセスできます。スレッドはプログラムのオブジェクトにアクセスするがあり、プログラマはスレッドでオブジェクトへのみきアクセスがしくされていることをするがあります。

StackOverflowにするトピック

- [タイプ](#)
- [Executor](#)、[ExecutorService](#)、[Threads](#) [ブール](#)
- [Thread](#)と[Runnable](#)

Examples

なマルチスレッド

するタスクがあり、これらのタスクがすべてのタスクのにしないは、マルチスレッドをして、コンピュータがであればのプロセッサをしてこれらのタスクをすべてにできます。きなしたタスクがあれば、これによりプログラムのがくなります。

```
class CountAndPrint implements Runnable {

    private final String name;

    CountAndPrint(String name) {
        this.name = name;
    }

    /** This is what a CountAndPrint will do */
    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            System.out.println(this.name + ": " + i);
        }
    }

    public static void main(String[] args) {
        // Launching 4 parallel threads
        for (int i = 1; i <= 4; i++) {
            // `start` method will call the `run` method
            // of CountAndPrint in another thread
            new Thread(new CountAndPrint("Instance " + i)).start();
        }

        // Doing some others tasks in the main Thread
    }
}
```

```

        for (int i = 0; i < 10000; i++) {
            System.out.println("Main: " + i);
        }
    }
}

```

さまざまな `CountAndPrint` インスタンスの `run` メソッドのコードは、なでされます。サンプルのスニペットはのようになります。

```

Instance 4: 1
Instance 2: 1
Instance 4: 2
Instance 1: 1
Instance 1: 2
Main: 1
Instance 4: 3
Main: 2
Instance 3: 1
Instance 4: 4
...

```

プロデューサー/コンシューマー

- のな。JDK クラス `AtomicBoolean` と `BlockingQueue` はにされるため、なソリューションがされるがります。さまざまな [BlockingQueue](#) について Javadoc にいわせてください。なるをすると、[DelayQueue](#) や [Priority Queue](#) などのこののがにされるがあります。

```

public class Producer implements Runnable {

    private final BlockingQueue<ProducedData> queue;

    public Producer(BlockingQueue<ProducedData> queue) {
        this.queue = queue;
    }

    public void run() {
        int producedCount = 0;
        try {
            while (true) {
                producedCount++;
                //put throws an InterruptedException when the thread is interrupted
                queue.put(new ProducedData());
            }
        } catch (InterruptedException e) {
            // the thread has been interrupted: cleanup and exit
            producedCount--;
            //re-interrupt the thread in case the interrupt flag is needed higher up
            Thread.currentThread().interrupt();
        }
        System.out.println("Produced " + producedCount + " objects");
    }
}

public class Consumer implements Runnable {

    private final BlockingQueue<ProducedData> queue;

```



```

public Consumer(BlockingQueue<ProducedData> queue) {
    this.queue = queue;
}

public void run() {
    int consumedCount = 0;
    try {
        while (true) {
            //put throws an InterruptedException when the thread is interrupted
            ProducedData data = queue.poll(10, TimeUnit.MILLISECONDS);
            // process data
            consumedCount++;
        }
    } catch (InterruptedException e) {
        // the thread has been interrupted: cleanup and exit
        consumedCount--;
        //re-interrupt the thread in case the interrupt flag is needed higher up
        Thread.currentThread().interrupt();
    }
    System.out.println("Consumed " + consumedCount + " objects");
}
}

public class ProducerConsumerExample {
    static class ProducedData {
        // empty data object
    }

    public static void main(String[] args) throws InterruptedException {
        BlockingQueue<ProducedData> queue = new ArrayBlockingQueue<ProducedData>(1000);
        // choice of queue determines the actual behavior: see various BlockingQueue
        implementations

        Thread producer = new Thread(new Producer(queue));
        Thread consumer = new Thread(new Consumer(queue));

        producer.start();
        consumer.start();

        Thread.sleep(1000);
        producer.interrupt();
        Thread.sleep(10);
        consumer.interrupt();
    }
}

```

ThreadLocalの

Java ConcurrencyのなツールはThreadLocalです。これにより、のスレッドにのをつことができます。したがって、じコードがなるスレッドでされる、これらののをしません、スレッドごとにスレッドのをちます。

たとえば、これはサーブレットのをするコンテキストなどをするためによくされます。あなたはそのようなことをするかもしれません

```
private static final ThreadLocal<MyUserContext> contexts = new ThreadLocal<>();

public static MyUserContext getContext() {
    return contexts.get(); // get returns the variable unique to this thread
}

public void doGet(...) {
    MyUserContext context = magicGetContextFromRequest(request);
    contexts.put(context); // save that context to our thread-local - other threads
                          // making this call don't overwrite ours

    try {
        // business logic
    } finally {
        contexts.remove(); // 'ensure' removal of thread-local variable
    }
}
```

では、`MyUserContext` をすべてのメソッドにすわりに、にじて `MyServlet.getContext()` することができます。はもちろん、これはするのがあるをしますが、スレッドセーフであるため、このようなスコープのきいをすることにはくがありません。

ここでのなは、すべてのスレッドがその `contexts` コンテナにのスレッドローカルをつことです。されたエンリポイントからサーブレットがコンテキストをするように、あるいはサーブレットフィルタをするようにするなどにして、このコンテキストにすることができます。

CountDownLatch

CountDownLatch

のスレッドでされているのがするまで、1つまたはのスレッドをさせる。

1. `CountDownLatch` は、されたカウントでされます。
2. `await` メソッドは、 `countDown()` メソッドのびしによってのカウントがゼロになるまでブロックします。その、のすべてのスレッドがされ、そののすべての `wait` のびしがすぐにります。
3. これはワンショットです。カウントをリセットすることはできません。カウントをリセットするバージョンがなは、 `CyclicBarrier` をして `CyclicBarrier`。

な

```
public void await() throws InterruptedException
```

スレッドがされないう、ラッチがゼロまでカウントダウンするまでのスレッドをさせます。

```
public void countDown()
```

ラッチのカウントをデクリメントし、カウントがゼロになるとすべてのスレッドをしします。

```
import java.util.concurrent.*;
```

```

class DoSomethingInAThread implements Runnable {
    CountdownLatch latch;
    public DoSomethingInAThread(CountdownLatch latch) {
        this.latch = latch;
    }
    public void run() {
        try {
            System.out.println("Do some thing");
            latch.countDown();
        } catch (Exception err) {
            err.printStackTrace();
        }
    }
}

public class CountdownLatchDemo {
    public static void main(String[] args) {
        try {
            int numberOfThreads = 5;
            if (args.length < 1) {
                System.out.println("Usage: java CountdownLatchDemo numberOfThreads");
                return;
            }
            try {
                numberOfThreads = Integer.parseInt(args[0]);
            } catch (NumberFormatException ne) {

            }
            CountdownLatch latch = new CountdownLatch(numberOfThreads);
            for (int n = 0; n < numberOfThreads; n++) {
                Thread t = new Thread(new DoSomethingInAThread(latch));
                t.start();
            }
            latch.await();
            System.out.println("In Main thread after completion of " + numberOfThreads + "
threads");
        } catch (Exception err) {
            err.printStackTrace();
        }
    }
}

```

```

java CountdownLatchDemo 5
Do some thing
Do some thing
Do some thing
Do some thing
Do some thing
Do some thing
In Main thread after completion of 5 threads

```

1. `CountdownLatch` は、メインスレッドのカウンタが5でされます
2. メインスレッドは `await()` メソッドをし `await()` して `await()` ます。
3. `DoSomethingInAThread` 5つのインスタンスがされました。インスタンスは、 `countDown()` メソッドを `countDown()` でカウンタをしました。
4. カウンタがゼロになると、メインスレッドがします

Javaでは、レベルのロックがみまれています。 `synchronized` ブロックは、Javaオブジェクトをみみロックとしてできますつまり、すべてのJavaオブジェクトにけられたモニターがあるがあります。

なロックは、のグループにをします。それがをするのかをするには、 `synchronized` がにつをてみましよう

```
private static int t = 0;
private static Object mutex = new Object();

public static void main(String[] args) {
    ExecutorService executorService = Executors.newFixedThreadPool(400); // The high thread
    count is for demonstration purposes.
    for (int i = 0; i < 100; i++) {
        executorService.execute(() -> {
            synchronized (mutex) {
                t++;
                System.out.println(MessageFormat.format("t: {0}", t));
            }
        });
    }
    executorService.shutdown();
}
```

この、 `synchronized` ブロックでなければ、ののがしていました。のものはポストインクリメントでありそれはアトミックではありません、2のものは、のののスレッドがそれをするをたにtのをすることです。しかし、なロックをして、ここにはなく、にはので1100のがまれます。

Javaのなロックは *mutex* つまりロックです。とは、のスレッドがロックをした、のスレッドがロックをするにのロックをするのをつことをします。スレッドをスリープにするのあるは、ブロッキングとばれます。したがって、ロックのはブロックである。

Javaのなロックはリエントラントです。これは、スレッドがすでにしているロックをしようとする、ブロックされず、にされることをします。たとえば、のコードはびされたときにブロックされません。

```
public void bar(){
    synchronized(this){
        ...
    }
}
public void foo(){
    synchronized(this){
        bar();
    }
}
```

`synchronized` ブロックのに、 `synchronized` もあります。

のコードブロックは、バイトコードがなるようにえても、にです。

1. `synchronized` のブロック `this`

```
public void foo() {
    synchronized(this) {
        doStuff();
    }
}
```

2. synchronizedメソッド

```
public synchronized void foo() {
    doStuff();
}
```

staticメソッドのもです。

```
class MyClass {
    ...
    public static void bar() {
        synchronized(MyClass.class) {
            doSomeOtherStuff();
        }
    }
}
```

これとじがあります

```
class MyClass {
    ...
    public static synchronized void bar() {
        doSomeOtherStuff();
    }
}
```

アトミックは、アトミックののにスレッドがをまたはすることなく、「に」されるです。

いをえてみましょう。

```
private static int t = 0;

public static void main(String[] args) {
    ExecutorService executorService = Executors.newFixedThreadPool(400); // The high thread
    count is for demonstration purposes.
    for (int i = 0; i < 100; i++) {
        executorService.execute(() -> {
            t++;
            System.out.println(MessageFormat.format("t: {0}", t));
        });
    }
    executorService.shutdown();
}
```

この、2つのがあります。のは、ポストインクリメントがアトミックではないことです。それはの
でされていますし、に1をえ、をします。そのため、をすると、にt: 100がされないがあります
。つまり、2つのスレッドがにをし、インクリメントしてすることができます。つまり、tのを

10とし、スレッドはtをインクリメントしています。のスレッドは、tのを11にします。これは、2のスレッドがのスレッドのインクリメントをするにtのをするためです。

2のは、どのように々がtをしているかである。tのをしているとき、はこのスレッドのインクリメントののにのスレッドによってにされているがあります。

これらのをするために、 `java.util.concurrent.atomic.AtomicInteger` をします。これは、くのアトミックができるようになっています。

```
private static AtomicInteger t = new AtomicInteger(0);

public static void main(String[] args) {
    ExecutorService executorService = Executors.newFixedThreadPool(400); // The high thread
    count is for demonstration purposes.
    for (int i = 0; i < 100; i++) {
        executorService.execute(() -> {
            int currentT = t.incrementAndGet();
            System.out.println(MessageFormat.format("t: {0}", currentT));
        });
    }
    executorService.shutdown();
}
```

`AtomicInteger`の`incrementAndGet`メソッドは、にしたいをするので、のはされます。このでは、`println`びしのをめることがないため、これはをとするため、こののであるため、はです。こののは、`AtomicInteger`をしてにするをする

なデッドロックシステムの

デッドロックは、2つのするアクションがもうのアクションをするのをっているときにします。javaでは、オブジェクトにけられたロックが1つあります。のオブジェクトののスレッドによってにわれるをけるために、`synchronized`キーワードをすることができますが、すべてはコストがかかります。`synchronized`キーワードをとてすると、システムがデッドロックされたになることがあります。

2つのスレッドが1つのインスタンスでしているとみなし、スレッドを1と2とし、2つのリソースR1とR2があるとします。にR1をし、のためにR2もですが、SecondはR2をし、するためにR1がです。

t = 0でうように、

にR1があり、2にR2があります。すぐにR2をとています.2がR1をとています。このちはで、デッドロックにつながります。

```
public class Example2 {

    public static void main(String[] args) throws InterruptedException {
        final DeadLock dl = new DeadLock();
        Thread t1 = new Thread(new Runnable() {
```

```

        @Override
        public void run() {
            // TODO Auto-generated method stub
            dl.methodA();
        }
    });

    Thread t2 = new Thread(new Runnable() {

        @Override
        public void run() {
            // TODO Auto-generated method stub
            try {
                dl.method2();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    });
    t1.setName("First");
    t2.setName("Second");
    t1.start();
    t2.start();
}

}

class DeadLock {

    Object mLock1 = new Object();
    Object mLock2 = new Object();

    public void methodA() {
        System.out.println("methodA wait for mLock1 " + Thread.currentThread().getName());
        synchronized (mLock1) {
            System.out.println("methodA mLock1 acquired " +
Thread.currentThread().getName());
            try {
                Thread.sleep(100);
                method2();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }

    public void method2() throws InterruptedException {
        System.out.println("method2 wait for mLock2 " + Thread.currentThread().getName());
        synchronized (mLock2) {
            System.out.println("method2 mLock2 acquired " +
Thread.currentThread().getName());
            Thread.sleep(100);
            method3();
        }
    }

    public void method3() throws InterruptedException {
        System.out.println("method3 mLock1 " + Thread.currentThread().getName());
        synchronized (mLock1) {
            System.out.println("method3 mLock1 acquired " +
Thread.currentThread().getName());

```

```

    }
}
}

```

このプログラムの

```

methodA wait for mLock1 First
method2 wait for mLock2 Second
method2 mLock2 acquired Second
methodA mLock1 acquired First
method3 mLock1 Second
method2 wait for mLock2 First

```

をする

`Thread.sleep`は、のスレッドをされたをさせます。これは、アプリケーションまたはコンピュータシステムでされているのアプリケーションののスレッドがプロセッサをできるようにするなです。`Thread`クラスには、2つのオーバーロードされた`sleep`メソッドがあります。

スリープをミリにするもの

```

public static void sleep(long millis) throws InterruptedException

```

スリープをナノにするもの

```

public static void sleep(long millis, int nanos)

```

を1する

```

Thread.sleep(1000);

```

これは、オペレーティングシステムのカーネルのスケジューラにするヒントであることにすることがです。これは必ずしもではないかもしれませんが、によってはナノのパラメータおそらくもいミリにめられるをしないものもあります。

`try / catch`に`Thread.sleep`びしをみ、`InterruptedException`をすることをおめします。

1をしているときのみきの

たちがっているように、`synchronized`キーワードを使ってメソッドやブロックをにするがあります。しかし、たちのいくつかは、してのもうつのなをしないかもしれません`synchronized`と`volatile`キーワードをにコードのをるから、それはまた、みり/きみバリアをします。このみり/きみバリアとはですかをってこれについてしましょう

```

class Counter {

    private Integer count = 10;

```



```

public synchronized void incrementCount() {
    count++;
}

public Integer getCount() {
    return count;
}
}

```

スレッドAが`incrementCount()`を呼び出し、その後スレッドBが`getCount()`を呼び出しましょう。このシナリオでは、Bに`count`がされるというはありません。それでも`count`が10とされることがありますが、`count`されたがしてられないもあり`count`。

このをするには、Javaメモリモデルがハードウェアアーキテクチャとどのようにされるかをするがあります。Javaでは、スレッドにはのスタックがあります。このスタックには、メソッド呼び出しスタックとそのスレッドでされたローカルがまれています。マルチコアシステムでは、2つのスレッドが々のコアでにされているがあります。このようなシナリオでは、スレッドのスタックのがコアのレジスタ/キャッシュにあるがあります。スレッドであれば、オブジェクトは`synchronized`または`volatile` キーワードをしてアクセスされ、`synchronized` ブロックのにスレッドはそののローカルコピーをメインメモリとさせます。これにより、読み/書きバリアがされ、スレッドがそのオブジェクトののをしていることをします。

しかし、スレッドBがにアクセスをしていないので、々のには、`count`、それはののうえれるかもしれない`count` レジスタにされているのとBは、々はするがあり、カウントののがえることをするために、スレッドAからをすることはないかもしれ`getCount()` もにします。

```

public synchronized Integer getCount() {
    return count;
}

```

スレッドAが`count` すると、`Counter` インスタンスのロックがされ、読み書きバリアがされ、そのブロックでされたすべてのがメインメモリにフラッシュされます。に、スレッドBが`Counter` のインスタンスでロックをすると、スレッドBは読み書きバリアになり、メインメモリから`count` を読み、すべてのをします。

Thread A

Acquire lock

Increment 'count'

Release lock

Flush everything to
main memory

Updates its local copy
with main memory

Acq

Re

Re

じのは、`volatile`みり/きみにもされます。`volatile`にきむにされたすべてののはメインメモリにフラッシュされ、`volatile`のみみのすべてのみりはメインメモリからのものになります。

java.lang.Thread インスタンスの

Javaでスレッドをするな2つのがあります。には、スレッドをすることは、されるコードをくこととじくらいです。2つのアプローチは、そのコードをするがなります。

Javaでは、スレッドはオブジェクト `java.lang.Thread` またはそのサブクラスのインスタンスによってされま^す。だからのアプローチは、そのサブクラスをし、`run`メソッドをオーバーライドすることです。

スレッドをして、`java.lang.Thread`クラスとスレッドをし、スレッドのをします。

```
class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Thread running!");
        }
    }
}
```

```
}
```

するコードをすでにしているので、スレッドはのようになにできます。

```
MyThread t = new MyThread();
```

[Thread](#)クラスには、スレッドのとしてされるをけるコンストラクタもまれています。これはマルチスレッドプログラムをデバッグするににちます。

```
class MyThread extends Thread {
    public MyThread(String name) {
        super(name);
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Thread running! ");
        }
    }
}

MyThread t = new MyThread("Greeting Producer");
```

2のは、[java.lang.Runnable](#)とそのメソッド`run`をしてコードをすることです。[Thread](#)クラスをすると、のスレッドでそのメソッドをできます。これをするには、[Runnable](#)インタフェースのインスタンスをけるコンストラクタをしてスレッドをします。

```
Thread t = new Thread(aRunnable);
```

これは、`lambda`またはメソッドJava 8のみとみわせるとになります。

```
Thread t = new Thread(operator::hardWork);
```

スレッドのもできます。

```
Thread t = new Thread(operator::hardWork, "Pi operator");
```

にえば、のアプローチをすることなくできます。しかし、[な](#)は、をするといます。

の4つのコンストラクタのそれぞれについて、のパラメータとして[java.lang.ThreadGroup](#)のインスタンスをけるもあります。

```
ThreadGroup tg = new ThreadGroup("Operators");
Thread t = new Thread(tg, operator::hardWork, "PI operator");
```

[ThreadGroup](#)はのスレッドをします。[スレッド](#)のコンストラクタをして[スレッド グループ](#)に [スレッド](#)をすることができます。[ThreadGroup](#)をして、すべての[Thread](#)をにすることができます。

また、[Thread](#)は[ThreadGroup](#)からをることができます。

したがって、[スレッド](#)をsumarizeするには、のいずれかのパブリックコンストラクタをして[Thread](#)をします。

```
Thread()
Thread(String name)
Thread(Runnable target)
Thread(Runnable target, String name)
Thread(ThreadGroup group, String name)
Thread(ThreadGroup group, Runnable target)
Thread(ThreadGroup group, Runnable target, String name)
Thread(ThreadGroup group, Runnable target, String name, long stackSize)
```

のものは、しいスレッドのましいスタックサイズをすることができます。

くの、じプロパティまたはじバターンのスレッドをしてすると、コードのがなわれることがあります。これは、[java.util.concurrent.ThreadFactory](#)をできるときです。このインタフェースをすると、ファクトリパターンとそのメソッド[newThreadRunnable](#)をしてスレッドをするプロシージャをカプセルできます。

```
class WorkerFactory implements ThreadFactory {
    private int id = 0;

    @Override
    public Thread newThread(Runnable r) {
        return new Thread(r, "Worker " + id++);
    }
}
```

スレッド/スレッド

Javaスレッドにはりみフラグがあり、はfalseです。スレッドをすることは、に、そのフラグをtrueにすることだけです。そのスレッドでされているコードは、々フラグをチェックしてそのフラグをすることができます。コードはにすることもできます。しかし、なぜそれぞれのスレッドはそのようなフラグをっていますかのところ、スレッドにブールのフラグをすることは、なにいつでもをすることができます。さて、のスレッドがされたときになるいをするメソッドがあります。これらのメソッドはブロッキングメソッドとされます。これらは、スレッドをWAITINGまたはTIMED_WAITINGにするメソッドです。スレッドがこのにあるときにりみをうと、りみフラグがtrueにされているのではなく、されたスレッドでInterruptedExceptionがスローされ、スレッドはびRUNNABLEになります。ブロッキングメソッドをびすコードは、チェックであるため、InterruptedExceptionをするがあります。だから、その、りみはにそれをする、WAITをするをすることができます。らかのでしているメソッドえばIOのブロッキングがスレッドをにしないので、そのようなでりみにするわけではないことにしてください。に、ブロッキングメソッドにるつまり、にるりみフラグがされたスレッドはすぐにInterruptedExceptionをスローし、りみフラグはクリアされます。

これらのにも、Javaはりみになをりてません。コードは、きなようにりみをにすることができます。しかし、ほとんどの、はスレッドにをるためにされ、もいでをするがあります。しかし、からかなように、をするために、そのりみににするのはそのスレッドのコードです。スレッドをすることはコラボレーションです。スレッドがされると、のコードはスタックトレースのさまでレベルになるがあります。ほとんどのコードではブロッキングメソッドがびされず、スレッドのをにらせることができないほどタイムリーにします。ほとんどの、りみにすることになるコードは、ループがっていないり、またはループがするようにフラグがされるまで、ループのコードです。おそらくのタスクをするループつまり、にしけるループは、ループをするためにりみフラグをチェックするがあります。ループの、セマンティクスは、すべてのタスクをするにしなければならぬか、またはいくつかのタスクをのままにすることがあるがあります。ブロッキングメソッドをびすコードは、`InterruptedException`をするようにされます。にであれば、に`InterruptedException`をし、それをスローするとできます。このように、それはびしにしてブロッキングメソッドそのものになります。をできないは、りみフラグをするがあります。したがって、びしはスレッドをしたことをスタックにします。によっては、`InterruptedException`になくメソッドをさせるがあります。この、メソッドがするまでりみフラグのをらせるがあります。ローカルをするがあります。そのスレッドをする。

されたタスクのをするコードの

```
class TaskHandler implements Runnable {

    private final BlockingQueue<Task> queue;

    TaskHandler(BlockingQueue<Task> queue) {
        this.queue = queue;
    }

    @Override
    public void run() {
        while (!Thread.currentThread().isInterrupted()) { // check for interrupt flag, exit
loop when interrupted
            try {
                Task task = queue.take(); // blocking call, responsive to interruption
                handle(task);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt(); // cannot throw InterruptedException (due
to Runnable interface restriction) so indicating interruption by setting the flag
            }
        }

        private void handle(Task task) {
            // actual handling
        }
    }
}
```

にするまでりみフラグのをらせるコードの

```
class MustFinishHandler implements Runnable {

    private final BlockingQueue<Task> queue;
```

```

MustFinishHandler(BlockingQueue<Task> queue) {
    this.queue = queue;
}

@Override
public void run() {
    boolean shouldInterrupt = false;

    while (true) {
        try {
            Task task = queue.take();
            if (task.isEndOfTasks()) {
                if (shouldInterrupt) {
                    Thread.currentThread().interrupt();
                }
                return;
            }
            handle(task);
        } catch (InterruptedException e) {
            shouldInterrupt = true; // must finish, remember to set interrupt flag when
we're done
        }
    }

    private void handle(Task task) {
        // actual handling
    }
}

```

タスクのリストをつが、されるとにするコードの

```

class GetAsFarAsPossible implements Runnable {

    private final List<Task> tasks = new ArrayList<>();

    @Override
    public void run() {
        for (Task task : tasks) {
            if (Thread.currentThread().isInterrupted()) {
                return;
            }
            handle(task);
        }

        private void handle(Task task) {
            // actual handling
        }
    }
}

```

グローバルキューをつのプロデューサ/コンシューマの

のコードは、のプロデューサー/コンシューマープログラムをしています。プロデューサスレッドとコンシューマスレッドはじグローバルキューをします。

```
import java.util.concurrent.*;
```

```

import java.util.Random;

public class ProducerConsumerWithES {
    public static void main(String args[]) {
        BlockingQueue<Integer> sharedQueue = new LinkedBlockingQueue<Integer>();

        ExecutorService pes = Executors.newFixedThreadPool(2);
        ExecutorService ces = Executors.newFixedThreadPool(2);

        pes.submit(new Producer(sharedQueue, 1));
        pes.submit(new Producer(sharedQueue, 2));
        ces.submit(new Consumer(sharedQueue, 1));
        ces.submit(new Consumer(sharedQueue, 2));

        pes.shutdown();
        ces.shutdown();
    }
}

/* Different producers produces a stream of integers continuously to a shared queue,
which is shared between all Producers and consumers */

class Producer implements Runnable {
    private final BlockingQueue<Integer> sharedQueue;
    private int threadNo;
    private Random random = new Random();
    public Producer(BlockingQueue<Integer> sharedQueue,int threadNo) {
        this.threadNo = threadNo;
        this.sharedQueue = sharedQueue;
    }
    @Override
    public void run() {
        // Producer produces a continuous stream of numbers for every 200 milli seconds
        while (true) {
            try {
                int number = random.nextInt(1000);
                System.out.println("Produced:" + number + ":by thread:"+ threadNo);
                sharedQueue.put(number);
                Thread.sleep(200);
            } catch (Exception err) {
                err.printStackTrace();
            }
        }
    }
}

/* Different consumers consume data from shared queue, which is shared by both producer and
consumer threads */
class Consumer implements Runnable {
    private final BlockingQueue<Integer> sharedQueue;
    private int threadNo;
    public Consumer (BlockingQueue<Integer> sharedQueue,int threadNo) {
        this.sharedQueue = sharedQueue;
        this.threadNo = threadNo;
    }
    @Override
    public void run() {
        // Consumer consumes numbers generated from Producer threads continuously
        while(true){
            try {
                int num = sharedQueue.take();
                System.out.println("Consumed: " + num + ":by thread:"+threadNo);
            }
        }
    }
}

```

```

        } catch (Exception err) {
            err.printStackTrace();
        }
    }
}
}

```

```

Produced:69:by thread:2
Produced:553:by thread:1
Consumed: 69:by thread:1
Consumed: 553:by thread:2
Produced:41:by thread:2
Produced:796:by thread:1
Consumed: 41:by thread:1
Consumed: 796:by thread:2
Produced:728:by thread:2
Consumed: 728:by thread:1

```

々

1. `sharedQueue` は、 `LinkedBlockingQueue` あり、すべてのプロデューサとコンシューマのスレッドでされます。
2. プロデューサスレッドは、200ミリごとに1つのをに `sharedQueue` し、 `sharedQueue` します
3. Consumer スレッドは `sharedQueue` からをに `Consumer` ます。
4. このプログラムは、に `synchronized` されたまたは `Lock` なしでされます。 `BlockingQueue` はそれをするためのです。

`BlockingQueue` は、にプロデューサ/コンシューマキューできるようにされています。

`BlockingQueue` のはスレッドセーフです。すべてのキューイングは、ロックまたはのをして、そのをアトミックにします。

みしアクセス

あるプロセスが "データ" をにみきするがあることがあります。

`ReadWriteLock` インタフェースとその `ReentrantReadWriteLock` では、のようにできるアクセスパターンをできます。

1. ののデータをにみることができます。なくとも1つのリーダアクセスがされている、ライターアクセスはです。
2. くの、1つのライターがデータにアクセスできます。ライターアクセスがされている、リーダはデータにアクセスできません。

はのようになります。

```

import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;
public class Sample {

```



```
// Our lock. The constructor allows a "fairness" setting, which guarantees the chronology of
lock attributions.
protected static final ReadWriteLock RW_LOCK = new ReentrantReadWriteLock();

// This is a typical data that needs to be protected for concurrent access
protected static int data = 0;

/** This will write to the data, in an exclusive access */
public static void writeToData() {
    RW_LOCK.writeLock().lock();
    try {
        data++;
    } finally {
        RW_LOCK.writeLock().unlock();
    }
}

public static int readData() {
    RW_LOCK.readLock().lock();
    try {
        return data;
    } finally {
        RW_LOCK.readLock().unlock();
    }
}
}
```

1 このユースケースは、`AtomicInteger` をしたよりクリーンなソリューションを `AtomicInteger` いますが、ここではするのはアクセスパターンです。これは、ここでのデータがであるというになくします。

2 のロックはにですが、カジュアルリーダーにはえないかもしれません。、あなたがでロックしていなければ、いくつかのことがっているがあります

1. プリミティブのきみは、すべてのJVMでアトミックであることがされていないため、`data` が64ビットのlongのは、64ビットきみの32ビットのみがみられ`data`
2. それをしなかったスレッドからのきみのは、きみとみりのに *Happen Before* のをしたにのみ、JVMによってされます。このは、リーダーとライタのがそれぞれのロックをするにされますが、そうでなければ

Java SE 8

よりいパフォーマンスがな、のタイプののでは、`StampedLock` とばれるよりいロックタイプがあります。これは、`StampedLock`、オブティミスティックロックモードをします。このロックは `ReadWriteLock` とはになるきをしますが、このサンプルはではありません。

オブジェクト

`Runnable` インタフェースはスレッドでされるコードをするためのメソッド `run()` します。

`Runnable` オブジェクトは `Thread` コンストラクタにされます。スレッドの `start()` メソッドがびされます。

```
public class HelloRunnable implements Runnable {

    @Override
    public void run() {
        System.out.println("Hello from a thread");
    }

    public static void main(String[] args) {
        new Thread(new HelloRunnable()).start();
    }
}
```

Java8の

```
public static void main(String[] args) {
    Runnable r = () -> System.out.println("Hello world");
    new Thread(r).start();
}
```

スレッドとスレッドサブクラス

`Runnable`ため、オブジェクトのは、よりである`Runnable`オブジェクトのクラスのサブクラスできる`Thread`。

`Thread`サブクラスはなアプリケーションではいやすくなりますが、タスククラスが`Thread`でなければならないという`Thread`ます。

`Runnable`オブジェクトは、レベルのスレッドAPIにできます。

セマフォ

セマフォは、スレッドによっておよびされるのをするシンクロナイザです。セマフォは、スレッドがするときにし、スレッドがされたときにするのカウンタとしてすることができます。スレッドがをみるときにのが0、がになるまでまたはスレッドがされるまでスレッドはブロックされます。

セマフォはのようにされます。

```
Semaphore semaphore = new Semaphore(1); // The int value being the number of permits
```

セマフォのコンストラクターは、のためにのブール・パラメーターをくれます。falseにすると、このクラスはスレッドがををしません。フェアネスがtrueにされている、セマフォは、`acquire`メソッドのいずれかをびすスレッドが、それらのメソッドのびしがされたでをるためにされることをします。それはのようにされます

```
Semaphore semaphore = new Semaphore(1, true);
```

に、javadocのをてみましょう。セマフォをってのプールへのアクセスをします。このでは、セマフォをして、`getItem()`がびされたときにされるがにすることをするために、ブロックをしています。

```

class Pool {
    /**
     * Note that this DOES NOT bound the amount that may be released!
     * This is only a starting value for the Semaphore and has no other
     * significant meaning UNLESS you enforce this inside of the
     * getNextAvailableItem() and markAsUnused() methods
     */
    private static final int MAX_AVAILABLE = 100;
    private final Semaphore available = new Semaphore(MAX_AVAILABLE, true);

    /**
     * Obtains the next available item and reduces the permit count by 1.
     * If there are no items available, block.
     */
    public Object getItem() throws InterruptedException {
        available.acquire();
        return getNextAvailableItem();
    }

    /**
     * Puts the item into the pool and add 1 permit.
     */
    public void putItem(Object x) {
        if (markAsUnused(x))
            available.release();
    }

    private Object getNextAvailableItem() {
        // Implementation
    }

    private boolean markAsUnused(Object o) {
        // Implementation
    }
}

```

スレッドプールをって2つの`int`をする

スレッドプールにはタスクのキューがあり、そのうちそれぞれがこれらのスレッドでされます。

のは、スレッドプールをして2つの`int`をするをしています。

Java SE 8

```

int[] firstArray = { 2, 4, 6, 8 };
int[] secondArray = { 1, 3, 5, 7 };
int[] result = { 0, 0, 0, 0 };

ExecutorService pool = Executors.newCachedThreadPool();

// Setup the ThreadPool:
// for each element in the array, submit a worker to the pool that adds elements
for (int i = 0; i < result.length; i++) {
    final int worker = i;
    pool.submit(() -> result[worker] = firstArray[worker] + secondArray[worker] );
}

// Wait for all Workers to finish:

```

```

try {
    // execute all submitted tasks
    pool.shutdown();
    // waits until all workers finish, or the timeout ends
    pool.awaitTermination(12, TimeUnit.SECONDS);
}
catch (InterruptedException e) {
    pool.shutdownNow(); //kill thread
}

System.out.println(Arrays.toString(result));

```

ノート

1. このはになものである。には、このさなタスクにスレッドをすることでスピードアップはありません。タスクのとスケジューリングのオーバーヘッドは、タスクをするのにやされるをするため、がされます。
2. Java 7をしていたは、タスクをするためにlambdaのわりにのクラスをします。

システムスレッドの、プログラムによってされたすべてのスレッドのステータスをする

コードスニペット

```

import java.util.Set;

public class ThreadStatus {
    public static void main(String args[]) throws Exception {
        for (int i = 0; i < 5; i++){
            Thread t = new Thread(new MyThread());
            t.setName("MyThread:" + i);
            t.start();
        }
        int threadCount = 0;
        Set<Thread> threadSet = Thread.getAllStackTraces().keySet();
        for (Thread t : threadSet) {
            if (t.getThreadGroup() == Thread.currentThread().getThreadGroup()) {
                System.out.println("Thread :" + t + ":" + "state:" + t.getState());
                ++threadCount;
            }
        }
        System.out.println("Thread count started by Main thread:" + threadCount);
    }
}

class MyThread implements Runnable {
    public void run() {
        try {
            Thread.sleep(2000);
        } catch (Exception err) {
            err.printStackTrace();
        }
    }
}

```

```
Thread :Thread[MyThread:1,5,main]:state:TIMED_WAITING
Thread :Thread[MyThread:3,5,main]:state:TIMED_WAITING
Thread :Thread[main,5,main]:state:RUNNABLE
Thread :Thread[MyThread:4,5,main]:state:TIMED_WAITING
Thread :Thread[MyThread:0,5,main]:state:TIMED_WAITING
Thread :Thread[MyThread:2,5,main]:state:TIMED_WAITING
Thread count started by Main thread:6
```

`Thread.getAllStackTraces().keySet()` は、アプリケーションスレッドとシステムスレッドをむすべての `Thread` します。アプリケーションがしたスレッドのだけにあるは、のスレッドのスレッドグループをメインプログラムスレッドとして `Thread` セットをりします。

のスレッドグループがない、プログラムはのシステムスレッドのステータスをします。

```
Reference Handler
Signal Dispatcher
Attach Listener
Finalizer
```

びしと

`Runnable` は、のスレッドでされるコードをラップするをしますが、をすることができないというがあります。 `Runnable` のかららかのりをするのは、 `Runnable` のスコープでアクセスなにをすることです。

`Callable` は Java 5 で `Runnable` ピアとしてされました。 `Callable` はにじですが、 `run` するわりに `call` メソッドをとっているがなり `run`。 `call` メソッドは、をすのをち、チェックをスローすることもできます。

Callable タスクののは、 **Future** をしてタップすることができます

`Future` は、 `Callable` のをするソートのコンテナとえることができます。びしのはのスレッドででき、 `Future` のをタップしようとするするとブロックされ、になったでがされます。

びしなインタフェース

```
public interface Callable<V> {
    V call() throws Exception;
}
```

```
interface Future<V> {
    V get();
    V get(long timeout, TimeUnit unit);
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
}
```

Callable との

```

public static void main(String[] args) throws Exception {
    ExecutorService es = Executors.newSingleThreadExecutor();

    System.out.println("Time At Task Submission : " + new Date());
    Future<String> result = es.submit(new ComplexCalculator());
    // the call to Future.get() blocks until the result is available. So we are in for about a
    10 sec wait now
    System.out.println("Result of Complex Calculation is : " + result.get());
    System.out.println("Time At the Point of Printing the Result : " + new Date());
}

```

いをするCallable

```

public class ComplexCalculator implements Callable<String> {

    @Override
    public String call() throws Exception {
        // just sleep for 10 secs to simulate a lengthy computation
        Thread.sleep(10000);
        System.out.println("Result after a lengthy 10sec calculation");
        return "Complex Result"; // the result
    }
}

```

```

Time At Task Submission : Thu Aug 04 15:05:15 EDT 2016
Result after a lengthy 10sec calculation
Result of Complex Calculation is : Complex Result
Time At the Point of Printing the Result : Thu Aug 04 15:05:25 EDT 2016

```

にされるの

`get()` はのをするメソッドですが、はプロビジョニングがあります

- `get(long timeout, TimeUnit unit)` は、のスレッドがをつをします。
- タスクコール `cancel(mayInterruptIfRunning)` をキャンセルします。フラグ `mayInterrupt` は、タスクがされ、されているにされるべきであることをす。
- `isDone()` びすことによって、タスクが/したかどうかをチェックします。
- いタスクがキャンセルされたかどうかをするには、 `isCancelled()` ます。

としてのロック

Java 5のパッケージは、スレッディングがレベルでした。このパッケージのにより、いくつかのよりいレベルのプログラミング/がされました。

ロックは、にブロックまたはキーワードとじをたすスレッドメカニズムです。

みみロック

```

int count = 0; // shared among multiple threads

public void doSomething() {
    synchronized(this) {

```

```

        ++count; // a non-atomic operation
    }
}

```

ロックをした

```

int count = 0; // shared among multiple threads

Lock lockObj = new ReentrantLock();
public void doSomething() {
    try {
        lockObj.lock();
        ++count; // a non-atomic operation
    } finally {
        lockObj.unlock(); // sure to release the lock without fail
    }
}

```

また、ロックは、ロックなどのなロックがしないが、にしたままであるか、ロックしようとするをし、なにはブロックしない。

にしてロックする

```

class Locky {
    int count = 0; // shared among multiple threads

    Lock lockObj = new ReentrantLock();

    public void doSomething() {
        try {
            try {
                lockObj.lockInterruptibly();
                ++count; // a non-atomic operation
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt(); // stopping
            }
        } finally {
            if (!Thread.currentThread().isInterrupted()) {
                lockObj.unlock(); // sure to release the lock without fail
            }
        }
    }
}

```

ロックできるときにのみかをする

```

public class Locky2 {
    int count = 0; // shared among multiple threads

    Lock lockObj = new ReentrantLock();

    public void doSomething() {
        boolean locked = lockObj.tryLock(); // returns true upon successful lock
        if (locked) {
            try {
                ++count; // a non-atomic operation
            }
        }
    }
}

```

```
        } finally {
            lockObj.unlock(); // sure to release the lock without fail
        }
    }
}
```

なロックにはいくつかのバリエーションがあります。については、 [こちらのAPIドキュメント](#)をしてください。

オンラインでプログラミングスレッドをむ <https://riptutorial.com/ja/java/topic/121/プログラミング-スレッド->

139:

にランダムなものはもないので、javadocはこれらのをとびます。これらはジェネレータでされます。

Examples

Javaは`utils`パッケージのとして、に`Random`というのジェネレータをします。このオブジェクトは、のみデータ `int`、`float` などとしてをするためにできます。また、これをして、ランダムなbool、またはランダムなバイトをすることもできます。はのとおります。

```
import java.util.Random;

...

Random random = new Random();
int randInt = random.nextInt();
long randLong = random.nextLong();

double randDouble = random.nextDouble(); //This returns a value between 0.0 and 1.0
float randFloat = random.nextFloat(); //Same as nextDouble

byte[] randBytes = new byte[16];
random.nextBytes(randBytes); //nextBytes takes a user-supplied byte array, and fills it with
random bytes. It returns nothing.
```

このクラスは、かなりのしかしません。などのにはしないでください。のランダムがなは、`SecureRandom`クラスをします。にべるように。「」と「でない」ランダムのについては、こののです。

の

`Random.nextInt(int bound)`メソッドは、の、つまりされるがよりさいをけります。ただし、`nextInt`メソッドだけがバインドをけれます。`nextLong`、`nextDouble`などしないでください。

```
Random random = new Random();
random.nextInt(1000); // 0 - 999

int number = 10 + random.nextInt(100); // number is in the range of 10 to 109
```

Java 1.7 `ThreadLocalRandom` [source](#) をすることもできます。このクラスは、スレッドセーフPRNGジェネレータをします。このクラスの`nextInt`メソッドは、とのをけることにしてください。

```
import java.util.concurrent.ThreadLocalRandom;

// nextInt is normally exclusive of the top value,
// so add 1 to make it inclusive
ThreadLocalRandom.current().nextInt(min, max + 1);
```

では、`bound`が $2^{30} + 1$ に、`nextInt(int bound)`はなことをうことができることにしてください。

アルゴリズムはややこしい。これは、 2^{31} が n で割り切れないというのために n をもたらしをする。がされるは n にする。のケースは、 $n = 2^{30} + 1$ であり、それにして、のは $1/2$ であり、ループのは 2 である。

つまり、バインディングをすると`nextInt`メソッドのパフォーマンスがしますが、このパフォーマンスのは、`bound`が`max int`のにつくほどになります。

になの

`Random`と`ThreadLocalRandom`は、々のはありますが、きながあります。[ジェネレータ](#)は、がかなりにできるアルゴリズムです。したがって、これらの2つのクラスは、のキーなどにはしていません。

がになPRNGがなで`java.security.SecureRandom`をすることができます。このクラスのインスタンスによってされたをすることは、クラスをになものとしてラベルけするほどしいです。

```
import java.security.SecureRandom;
import java.util.Arrays;

public class Foo {
    public static void main(String[] args) {
        SecureRandom rng = new SecureRandom();
        byte[] randomBytes = new byte[64];
        rng.nextBytes(randomBytes); // Fills randomBytes with random bytes (duh)
        System.out.println(Arrays.toString(randomBytes));
    }
}
```

`SecureRandom`は、にであることにえて、 2^{48} の`Random`として、 2^{160} というなをっています。これは、よりかなりいことのつのがある`Random`とのようなののPRNG [メルセンヌツイスター](#)と[Xorshift](#)をしかし、。

`SecureRandom`は、プラットフォームとプロバイダのにします。デフォルトの`SecureRandom` `sun.security.provider.SecureRandom` `SUN` プロバイダによってされ`sun.security.provider.SecureRandom`

- Unixのようなシステムでは、`/dev/random`や`/dev/urandom`からのデータがシードされています。
- Windowsでは、[CryptoAPI](#)の`CryptGenRandom()`がシードされています。

のないをする

```
/**
 * returns a array of random numbers with no duplicates
 * @param range the range of possible numbers for ex. if 100 then it can be anywhere from 1-100
 * @param length the length of the array of random numbers
```

```

* @return array of random numbers with no duplicates.
*/
public static int[] getRandomNumbersWithNoDuplicates(int range, int length){
    if (length<range){
        // this is where all the random numbers
        int[] randomNumbers = new int[length];

        // loop through all the random numbers to set them
        for (int q = 0; q < randomNumbers.length; q++){

            // get the remaining possible numbers
            int remainingNumbers = range-q;

            // get a new random number from the remainingNumbers
            int newRandSpot = (int) (Math.random()*remainingNumbers);

            newRandSpot++;

            // loop through all the possible numbers
            for (int t = 1; t < range+1; t++){

                // check to see if this number has already been taken
                boolean taken = false;
                for (int number : randomNumbers){
                    if (t==number){
                        taken = true;
                        break;
                    }
                }

                // if it hasnt been taken then remove one from the spots
                if (!taken){
                    newRandSpot--;

                    // if we have gone though all the spots then set the value
                    if (newRandSpot==0){
                        randomNumbers[q] = t;
                    }
                }
            }
        }
        return randomNumbers;
    } else {
        // invalid can't have a length larger then the range of possible numbers
    }
    return null;
}

```

このメソッドは、されたさのサイズをつをループし、りのさをつけることによってします。これらののをnewRandSpotにし、そののをのにつけます。これは、をループし、そのがすでにされているかどうかをすることによってこれをいます。

たとえば、が5でさが3で、すでに2をしているはりの4つのがあるので、1と4のをし、のをスキップする5をループしますたちはすでにしています2。

は、1と4のでされたのが3であるとしましょう。のループでは、まだりされていない1がられます

。3から1をりくことができます。2.は2のループで、たちはもしません。たちは4になるまでこのパターンにいます.1をすると0になりますので、しいrandomNumberを4にします。

されたシードでをする

```
//Creates a Random instance with a seed of 12345.
Random random = new Random(12345L);

//Gets a ThreadLocalRandom instance
ThreadLocalRandom tlr = ThreadLocalRandom.current();

//Set the instance's seed.
tlr.setSeed(12345L);
```

じシードをしてをすると、じがされるため、したになりたくないは、すべてのRandomインスタンスにしてなるシードをすることをおめします。

すべてのびしでなるLongをするいは、`System.currentTimeMillis()`です。

```
Random random = new Random(System.currentTimeMillis());
ThreadLocalRandom.current().setSeed(System.currentTimeMillis());
```

apache-common lang3をとてをする

`org.apache.commons.lang3.RandomUtils`をして、1でをすることができます。

```
int x = RandomUtils.nextInt(1, 1000);
```

`nextInt(int startInclusive, int endExclusive)`メソッドはをとります。

`int`とはに、このクラスをして `long`、`double`、`float` および `bytes` をランダムにでき `bytes`。

`RandomUtils`クラスにはのメソッドがまれています。

```
static byte[] nextBytes(int count) //Creates an array of random bytes.
static double nextDouble() //Returns a random double within 0 - Double.MAX_VALUE
static double nextDouble(double startInclusive, double endInclusive) //Returns a random double
within the specified range.
static float nextFloat() //Returns a random float within 0 - Float.MAX_VALUE
static float nextFloat(float startInclusive, float endInclusive) //Returns a random float
within the specified range.
static int nextInt() //Returns a random int within 0 - Integer.MAX_VALUE
static int nextInt(int startInclusive, int endExclusive) //Returns a random integer within the
specified range.
static long nextLong() //Returns a random long within 0 - Long.MAX_VALUE
static long nextLong(long startInclusive, long endExclusive) //Returns a random long within
the specified range.
```

オンラインでをむ <https://riptutorial.com/ja/java/topic/890/>

140: コレクション

このトピックでは、guava、apache、eclipseMultiset、Bag、Multimap、utilsなどのJavaコレクションについてします。

Examples

Apache HashBag、Guava HashMultisetおよびEclipse HashBag

Bag / ultisetは、オブジェクトをコレクションのオカレンスのとともにします。インターフェイスのなメソッドにより、オブジェクトののコピーをにまたはすることができます。がこのキーのコピーである、JDKのanalogはHashMap <T、Integer>です。

タイプ	グアバ	Apache Commons Collections	GSコレクション	JDK
が さ れ て い な い	HashMultiset	HashBag	HashBag	ハッシュマップ
ソ ー ト み	TreeMultiset	TreeBag	TreeBag	ツリーマップ
	LinkedHashMultiset	-	-	リンクされたハッシュ
バ リ ア ン ト	ConcurrentHashMultiset	SynchronizedBag	SynchronizedBag	Collections.synchron
ソ ー ト	-	SynchronizedSortedBag	SynchronizedSortedBag	Collections.synchron
の	マルチセット	UnmodifiableBag	UnmodifiableBag	Collections.unmodif

タイプ	グアバ	Apache Commons Collections	GSコレクション	JDK
コレクション				
でソートされた	のソートされたマルチセット	UnmodifiableSortedBag	UnmodifiableSortedBag	Collections.unmodifiableSortedBag

1. ApacheのSynchronizedSortedBagの

```
// Parse text to separate words
String INPUT_TEXT = "Hello World! Hello All! Hi World!";
// Create Multiset
Bag bag = SynchronizedSortedBag.synchronizedBag(new
TreeBag(Arrays.asList(INPUT_TEXT.split(" "))));

// Print count words
System.out.println(bag); // print [1:All!,2:Hello,1:Hi,2:World!]- in natural (alphabet)
order
// Print all unique words
System.out.println(bag.uniqueSet()); // print [All!, Hello, Hi, World!]- in natural
(alphabet) order

// Print count occurrences of words
System.out.println("Hello = " + bag.getCount("Hello")); // print 2
System.out.println("World = " + bag.getCount("World!")); // print 2
System.out.println("All = " + bag.getCount("All!")); // print 1
System.out.println("Hi = " + bag.getCount("Hi")); // print 1
System.out.println("Empty = " + bag.getCount("Empty")); // print 0

// Print count all words
System.out.println(bag.size()); //print 6

// Print count unique words
System.out.println(bag.uniqueSet().size()); //print 4
```

EclipseGCのTreeBagをする

```
// Parse text to separate words
```

```

String INPUT_TEXT = "Hello World! Hello All! Hi World!";
// Create Multiset
MutableSortedBag<String> bag = TreeBag.newBag(Arrays.asList(INPUT_TEXT.split(" ")));

// Print count words
System.out.println(bag); // print [All!, Hello, Hello, Hi, World!, World!]- in natural
order
// Print all unique words
System.out.println(bag.toSortedSet()); // print [All!, Hello, Hi, World!]- in natural
order

// Print count occurrences of words
System.out.println("Hello = " + bag.occurrencesOf("Hello")); // print 2
System.out.println("World = " + bag.occurrencesOf("World!")); // print 2
System.out.println("All = " + bag.occurrencesOf("All!")); // print 1
System.out.println("Hi = " + bag.occurrencesOf("Hi")); // print 1
System.out.println("Empty = " + bag.occurrencesOf("Empty")); // print 0

// Print count all words
System.out.println(bag.size()); //print 6

// Print count unique words
System.out.println(bag.toSet().size()); //print 4

```

3. GuavaのLinkedHashMultisetをする

```

// Parse text to separate words
String INPUT_TEXT = "Hello World! Hello All! Hi World!";
// Create Multiset
Multiset<String> multiset = LinkedHashMultiset.create(Arrays.asList(INPUT_TEXT.split("
")));

// Print count words
System.out.println(multiset); // print [Hello x 2, World! x 2, All!, Hi]- in predictable
iteration order
// Print all unique words
System.out.println(multiset.elementSet()); // print [Hello, World!, All!, Hi] - in
predictable iteration order

// Print count occurrences of words
System.out.println("Hello = " + multiset.count("Hello")); // print 2
System.out.println("World = " + multiset.count("World!")); // print 2
System.out.println("All = " + multiset.count("All!")); // print 1
System.out.println("Hi = " + multiset.count("Hi")); // print 1
System.out.println("Empty = " + multiset.count("Empty")); // print 0

// Print count all words
System.out.println(multiset.size()); //print 6

// Print count unique words
System.out.println(multiset.elementSet().size()); //print 4

```

そのの

1. Apacheコレクション

1. [HashBag](#) - されていない
2. [SynchronizedBag](#) - とがされていない
3. [SynchronizedSortedBag](#) - ソート
4. [TreeBag](#) - ソート

II。GS / Eclipseコレクション

5. [MutableBag](#) - されていない
6. [MutableSortedBag](#) - ソート

III。グアバ

7. [HashMultiset](#) - されていないオーダー
8. [TreeMultiset](#) - ソート
9. [LinkedHashMultiset](#) - オーダー
10. [ConcurrentHashMultiset](#) - とがされていない

Guava、ApacheおよびEclipseコレクションのマルチマップ

このマルチマップでは、するキーとのペアをできます。JDKのは、`HashMap <K、List>`、`HashMap <K、Set>`などです。

キーの	の		アナログキー	アナログ	グアバ	Apache	EclipseGSコレクション
されていない		はい	ハッシュマップ	リスト	ArrayListMultimap	<code>MultiValueMap</code>	<code>FastListMulti</code>
されていない	されていない	はい	ハッシュマップ	HashSet	HashMultimap	<code>MultiValueMap. multiValueMap(new HashMap<K, Set>(), HashSet.class);</code>	<code>UnifiedSetMul</code>

キーの	の		アナログキー	アナログ	グアバ	Apache	EclipseGSコ シヨン
されて います ません	べえ	いいえ	ハ ッ シ ユ マ ッ プ	TreeSet	<pre> Multimaps. newMultimap(HashMap, Supplier <TreeSet>)</pre>	<pre> MultiValueMap.multiValueMap(new HashMap<K, Set>(), TreeSet.class)</pre>	TreeSortedSet Multimap
		はい	リ ン ク さ れ た ハ ッ シ ユ マ ッ プ	リスト	LinkedListMultimap	MultiValueMap。 multiValueMapしい LinkedHashMap <K、 List> 、 ArrayList.class;	
		いいえ	リ ン ク さ れ た ハ ッ シ ユ マ ッ プ	LinkedHashSet	LinkedHashMultimap	<pre> MultiValueMap. multiValueMap(new LinkedHashMap<K, Set>(), LinkedHashSet.class)</pre>	
べ	べ	い	ッ	TreeSet	TreeMultimap	<pre> MultiValueMap. multiValueMap(new</pre>	

キー の	の		アナ ログ キー	アナログ	グアバ	Apache	EclipseGSコ レクション
え	え	い え	リ ー マ ッ プ			TreeMap<K, Set>(),TreeSet.class)	

マルチマップをした

タスク「Hello WorldHello AllHi World」をするをし、MultiMapをしてすべてののすべてのインデックスをしますたとえば、Hello = [0、2]、World= [1,5]など

1. ApacheのMultiValueMap

```
String INPUT_TEXT = "Hello World! Hello All! Hi World!";
// Parse text to words and index
List<String> words = Arrays.asList(INPUT_TEXT.split(" "));
// Create Multimap
MultiMap<String, Integer> multiMap = new MultiValueMap<String, Integer>();

// Fill Multimap
int i = 0;
for(String word: words) {
    multiMap.put(word, i);
    i++;
}

// Print all words
System.out.println(multiMap); // print {Hi=[4], Hello=[0, 2], World!=[1, 5], All!=[3]} -
in random orders
// Print all unique words
System.out.println(multiMap.keySet()); // print [Hi, Hello, World!, All!] - in random
orders

// Print all indexes
System.out.println("Hello = " + multiMap.get("Hello")); // print [0, 2]
System.out.println("World = " + multiMap.get("World!")); // print [1, 5]
System.out.println("All = " + multiMap.get("All!")); // print [3]
System.out.println("Hi = " + multiMap.get("Hi")); // print [4]
System.out.println("Empty = " + multiMap.get("Empty")); // print null

// Print count unique words
System.out.println(multiMap.keySet().size()); //print 4
```

2. GS / EclipseコレクションのHashBiMap

```
String[] englishWords = {"one", "two", "three","ball","snow"};
```

```

String[] russianWords = {"jeden", "dwa", "trzy", "kula", "snieg"};

// Create Multiset
MutableBiMap<String, String> biMap = new HashBiMap(englishWords.length);
// Create English-Polish dictionary
int i = 0;
for(String englishWord: englishWords) {
    biMap.put(englishWord, russianWords[i]);
    i++;
}

// Print count words
System.out.println(biMap); // print {two=dwa, ball=kula, one=jeden, snow=snieg,
three=trzy} - in random orders
// Print all unique words
System.out.println(biMap.keySet()); // print [snow, two, one, three, ball] - in random
orders
System.out.println(biMap.values()); // print [dwa, kula, jeden, snieg, trzy] - in
random orders

// Print translate by words
System.out.println("one = " + biMap.get("one")); // print one = jeden
System.out.println("two = " + biMap.get("two")); // print two = dwa
System.out.println("kula = " + biMap.inverse().get("kula")); // print kula = ball
System.out.println("snieg = " + biMap.inverse().get("snieg")); // print snieg = snow
System.out.println("empty = " + biMap.get("empty")); // print empty = null

// Print count word's pair
System.out.println(biMap.size()); //print 5

```

3. GuavaのHashMultiMap

```

String INPUT_TEXT = "Hello World! Hello All! Hi World!";
// Parse text to words and index
List<String> words = Arrays.asList(INPUT_TEXT.split(" "));
// Create Multimaps
Multimap<String, Integer> multiMap = HashMultimap.create();

// Fill Multimaps
int i = 0;
for(String word: words) {
    multiMap.put(word, i);
    i++;
}

// Print all words
System.out.println(multiMap); // print {Hi=[4], Hello=[0, 2], World!=[1, 5], All!=[3]} -
keys and values in random orders
// Print all unique words
System.out.println(multiMap.keySet()); // print [Hi, Hello, World!, All!] - in random
orders

// Print all indexes
System.out.println("Hello = " + multiMap.get("Hello")); // print [0, 2]
System.out.println("World = " + multiMap.get("World!")); // print [1, 5]
System.out.println("All = " + multiMap.get("All!")); // print [3]
System.out.println("Hi = " + multiMap.get("Hi")); // print [4]
System.out.println("Empty = " + multiMap.get("Empty")); // print []

// Print count all words

```

```
System.out.println(multiMap.size());    //print 6

// Print count unique words
System.out.println(multiMap.keySet().size());    //print 4
```

Noreの

I. Apacheコレクション

1. [MultiValueMap](#)
2. [MultiValueMapLinked](#)
3. [MultiValueMapTree](#)

II. GS / Eclipseコレクション

1. [FastListMultimap](#)
2. [HashBagMultimap](#)
3. [TreeSortedSetMultimap](#)
4. [UnifiedSetMultimap](#)

III. グアバ

1. [HashMultiMap](#)
2. [LinkedHashMultimap](#)
3. [LinkedListMultimap](#)
4. [TreeMultimap](#)
5. [ArrayListMultimap](#)

コレクションとをやる . コレクションをやる

コレクションとをやる . コレクションをやる

1. リストをやる

	JDK	グアバ	gs-collections
のリストをやる	<code>new ArrayList<></code>	<code>Lists.newArrayList()</code>	<code>FastList.newList()</code>
からリストをやる	<code>Arrays.asList("1", "2", "3")</code>	<code>Lists.newArrayList("1", "2", "3")</code>	<code>FastList.newListWith("1", "2", "3")</code>
= 100 のリストをやる	<code>new ArrayList<>(100)</code>	<code>Lists.newArrayListWithCapacity(100)</code>	<code>FastList.newList(100)</code>

	JDK	グアバ	gs-collections
の collection からリ ストを する	<code>new ArrayList<>(collection)</code>	<code>Lists.newArrayList(collection)</code>	<code>FastList.newList(collection)</code>
の Iterable からリ ストを する	-	<code>Lists.newArrayList(iterable)</code>	<code>FastList.newList(iterable)</code>
Iterator からリ ストを する	-	<code>Lists.newArrayList(iterator)</code>	-
からリ ストを する	<code>Arrays.asList(array)</code>	<code>Lists.newArrayList(array)</code>	<code>FastList.newListWith(array)</code>
をして リスト をする	-	-	<code>FastList.newWithNValues(10, () -> "1")</code>

```

System.out.println("createArrayList start");
// Create empty list
List<String> emptyGuava = Lists.newArrayList(); // using guava
List<String> emptyJDK = new ArrayList<>(); // using JDK
MutableList<String> emptyGS = FastList.newList(); // using gs

// Create list with 100 element
List<String> exactly100 = Lists.newArrayListWithCapacity(100); // using guava
List<String> exactly100JDK = new ArrayList<>(100); // using JDK
MutableList<String> empty100GS = FastList.newList(100); // using gs

// Create list with about 100 element
List<String> approx100 = Lists.newArrayListWithExpectedSize(100); // using guava
List<String> approx100JDK = new ArrayList<>(115); // using JDK
MutableList<String> approx100GS = FastList.newList(115); // using gs

// Create list with some elements
List<String> withElements = Lists.newArrayList("alpha", "beta", "gamma"); // using guava
List<String> withElementsJDK = Arrays.asList("alpha", "beta", "gamma"); // using JDK
MutableList<String> withElementsGS = FastList.newListWith("alpha", "beta", "gamma"); //
using gs

System.out.println(withElements);
System.out.println(withElementsJDK);
System.out.println(withElementsGS);

```

```

// Create list from any Iterable interface (any collection)
Collection<String> collection = new HashSet<>(3);
collection.add("1");
collection.add("2");
collection.add("3");

List<String> fromIterable = Lists.newArrayList(collection); // using guava
List<String> fromIterableJDK = new ArrayList<>(collection); // using JDK
MutableList<String> fromIterableGS = FastList.newList(collection); // using gs

System.out.println(fromIterable);
System.out.println(fromIterableJDK);
System.out.println(fromIterableGS);
/* Attention: JDK create list only from Collection, but guava and gs can create list from
Iterable and Collection */

// Create list from any Iterator
Iterator<String> iterator = collection.iterator();
List<String> fromIterator = Lists.newArrayList(iterator); // using guava
System.out.println(fromIterator);

// Create list from any array
String[] array = {"4", "5", "6"};
List<String> fromArray = Lists.newArrayList(array); // using guava
List<String> fromArrayJDK = Arrays.asList(array); // using JDK
MutableList<String> fromArrayGS = FastList.newListWith(array); // using gs
System.out.println(fromArray);
System.out.println(fromArrayJDK);
System.out.println(fromArrayGS);

// Create list using fabric
MutableList<String> fromFabricGS = FastList.newWithNValues(10, () ->
String.valueOf(Math.random())); // using gs
System.out.println(fromFabricGS);

System.out.println("createArrayList end");

```

2セットをする

	JDK	グアバ	gs-collections
のセットをする	<code>new HashSet<>()</code>	<code>Sets.newHashSet()</code>	<code>UnifiedSet.newSet()</code>
からされたク リエイ ティブ	<code>new HashSet<>(Arrays.asList("alpha", "beta", "gamma"))</code>	<code>Sets.newHashSet("alpha", "beta", "gamma")</code>	<code>UnifiedSet.newSetWith("alpha", "beta", "gamma")</code>
のコレ クショ ンから セット を	<code>new HashSet<>(collection)</code>	<code>Sets.newHashSet(collection)</code>	<code>UnifiedSet.newSet(collection)</code>

	JDK	グアバ	gs-collections
の Iterable からセ ットを する	-	Sets.newHashSet(iterable)	UnifiedSet.newSet(iterable)
のイテ レータ からセ ットを する	-	Sets.newHashSet(iterator)	-
からセ ットを する	new HashSet<>(Arrays.asList(array))	Sets.newHashSet(array)	UnifiedSet.newSetWith(array)

```

System.out.println("createHashSet start");
// Create empty set
Set<String> emptyGuava = Sets.newHashSet(); // using guava
Set<String> emptyJDK = new HashSet<>(); // using JDK
Set<String> emptyGS = UnifiedSet.newSet(); // using gs

// Create set with 100 element
Set<String> approx100 = Sets.newHashSetWithExpectedSize(100); // using guava
Set<String> approx100JDK = new HashSet<>(130); // using JDK
Set<String> approx100GS = UnifiedSet.newSet(130); // using gs

// Create set from some elements
Set<String> withElements = Sets.newHashSet("alpha", "beta", "gamma"); // using guava
Set<String> withElementsJDK = new HashSet<>(Arrays.asList("alpha", "beta", "gamma")); //
using JDK
Set<String> withElementsGS = UnifiedSet.newSetWith("alpha", "beta", "gamma"); // using gs

System.out.println(withElements);
System.out.println(withElementsJDK);
System.out.println(withElementsGS);

// Create set from any Iterable interface (any collection)
Collection<String> collection = new ArrayList<>(3);
collection.add("1");
collection.add("2");
collection.add("3");

Set<String> fromIterable = Sets.newHashSet(collection); // using guava
Set<String> fromIterableJDK = new HashSet<>(collection); // using JDK
Set<String> fromIterableGS = UnifiedSet.newSet(collection); // using gs

System.out.println(fromIterable);
System.out.println(fromIterableJDK);
System.out.println(fromIterableGS);
/* Attention: JDK create set only from Collection, but guava and gs can create set from
Iterable and Collection */

```

```
// Create set from any Iterator
Iterator<String> iterator = collection.iterator();
Set<String> fromIterator = Sets.newHashSet(iterator); // using guava
System.out.println(fromIterator);

// Create set from any array
String[] array = {"4", "5", "6"};
Set<String> fromArray = Sets.newHashSet(array); // using guava
Set<String> fromArrayJDK = new HashSet<>(Arrays.asList(array)); // using JDK
Set<String> fromArrayGS = UnifiedSet.newSetWith(array); // using gs
System.out.println(fromArray);
System.out.println(fromArrayJDK);
System.out.println(fromArrayGS);

System.out.println("createHashSet end");
```

3をする

	JDK	グアバ	gs-collections
の を す る	<code>new HashMap<>()</code>	<code>Maps.newHashMap()</code>	<code>UnifiedMap.newMap()</code>
= 130 の を す る	<code>new HashMap<>(130)</code>	<code>Maps.newHashMapWithExpectedSize(100)</code>	<code>UnifiedMap.newMap(130)</code>
の か ら を す る	<code>new HashMap<>(map)</code>	<code>Maps.newHashMap(map)</code>	<code>UnifiedMap.newMap(map)</code>
キ ー か ら マ ッ プ を す る	-	-	<code>UnifiedMap.newWithKeysValues("1", "a", "2", "b")</code>


```

System.out.println("createHashMap start");
// Create empty map
Map<String, String> emptyGuava = Maps.newHashMap(); // using guava
Map<String, String> emptyJDK = new HashMap<>(); // using JDK
Map<String, String> emptyGS = UnifiedMap.newMap(); // using gs

// Create map with about 100 element
Map<String, String> approx100 = Maps.newHashMapWithExpectedSize(100); // using guava
Map<String, String> approx100JDK = new HashMap<>(130); // using JDK
Map<String, String> approx100GS = UnifiedMap.newMap(130); // using gs

// Create map from another map
Map<String, String> map = new HashMap<>(3);
map.put("k1", "v1");
map.put("k2", "v2");
Map<String, String> withMap = Maps.newHashMap(map); // using guava
Map<String, String> withMapJDK = new HashMap<>(map); // using JDK
Map<String, String> withMapGS = UnifiedMap.newMap(map); // using gs

System.out.println(withMap);
System.out.println(withMapJDK);
System.out.println(withMapGS);

// Create map from keys
Map<String, String> withKeys = UnifiedMap.newWithKeysValues("1", "a", "2", "b");
System.out.println(withKeys);

System.out.println("createHashMap end");

```

そのの [CreateCollectionTest](#)

1. [CollectionCompare](#)
2. [コレクション](#)
3. [JavaTransform](#)

オンラインでコレクションをむ <https://riptutorial.com/ja/java/topic/2958/コレクション>

141: と

き

`Throwable`のオブジェクトとそのサブタイプは、`throw`キーワードでスタックにし、`try...catch`で
`try...catch`ことができます。

- `void someMethodthrows SomeException {}` //メソッド、`SomeException`がチェックされたのであるにメソッドびしににキャッチさせる

- `して{`

```
someMethod(); //code that might throw an exception
```

```
}
```

- `catchSomeException e{`

```
System.out.println("SomeException was thrown!"); //code that will run if certain  
exception (SomeException) is thrown
```

```
}
```

- `に{`

```
//code that will always run, whether try block finishes or not
```

```
}
```

Examples

`try-catch`をってをキャッチする

`try...catch`ステートメントをしてをキャッチしてできます。に`try`ステートメントは
[try...catch...finally](#)と[try-with-resources](#)にするのでされているように、のをとっています。

1つのキャッチブロックでしてみる

もシンプルなフォームはのようになります。

```
try {  
    doSomething();  
} catch (SomeException e) {  
    handle(e);  
}
```

```
}  
// next statement
```

な `try...catch` のはのとおりです。

- `try` ブロックのステートメントがされます。
- `try` ブロックのステートメントによってがスローされない、コントロールは `try...catch` あとにのステートメントにされ `try...catch` 。
- `try` ブロックにがスローされた
 - オブジェクトは、 `SomeException` またはサブタイプのインスタンスかどうかをべるためにテストされます。
 - そうであれば、 `catch` ブロックはをキャッチします
 - `e` はオブジェクトにバインドされています。
 - `catch` ブロックのコードがされます。
 - そのコードがをスローすると、しくスローされたがののわりにされます。
 - それのは、 `try...catch` にののが `try...catch` ます。
 - そうでない、のはききします。

のキャッチでしてみる

`try...catch` はの `catch` ブロックをつこともできます。えば

```
try {  
    doSomething();  
} catch (SomeException e) {  
    handleOneWay(e)  
}  
catch (SomeOtherException e) {  
    handleAnotherWay(e);  
}  
// next statement
```

の `catch` ブロックがある、の `catch` ブロックから1ずつされ、のがされます。するハンドラがのようになされ、 `try...catch` のののがされます。 ハンドラコードがをスローしたとしても、したの `catch` ブロックはにスキップされます。

「トップダウン」マッチングは、 `catch` ブロックのがいにしていないにします。えば

```
try {  
    throw new RuntimeException("test");  
} catch (Exception e) {  
    System.out.println("Exception");  
} catch (RuntimeException e) {  
    System.out.println("RuntimeException");  
}
```

このコードスニペットは、 "RuntimeException"ではなく "Exception"をします。 `RuntimeException` は `Exception` サブタイプであるため、のよりな `catch` がマッチします。 2のよりな `catch` はしてされません。

これからすべきは、タイプにしてもな `catch` ブロックがにれ、もな `catch` ブロックがにれるべきであるということです。のJavaコンパイラは、`catch` をできないにしますが、これはコンパイルエラーではありません。

マルチキャッチブロック

Java SE 7

Java SE 7、の `catch` ブロックでな のリストをできます。タイプが `|` でられてリストされます。例えば

```
try {
    doSomething();
} catch (SomeException | SomeOtherException e) {
    handleSomeException(e);
}
```

マルチキャッチのは、ののなです。 `catch` れたは、スローされたがリストされたのなくとも1つとするにします。

にはさらになものがあります。 `e` タイプは、リストのタイプのユニオンです。 `e` のがされるとき、そののはのなです。ただし、 `catch` ブロックで `e` がスローされた、スローされるのはののです。例えば

```
public void method() throws IOException, SQLException
{
    try {
        doSomething();
    } catch (IOException | SQLException e) {
        report(e);
        throw e;
    }
}
```

のでは、 `IOException` と `SQLException` は、のスーパータイプが `Exception` をし `Exception`。つまり、`report` メソッドは `report(Exception)` するがあり `report(Exception)`。ただし、コンパイラは `throw` が `IOException` または `SQLException` のみを `throw` できることをします。したがって、 `throws Exception` ではなく `throws IOException, SQLException` として `method` をでき `throws Exception`。これはいいことです [Pitfall - Throwable Throwable、Exception、Error、またはRuntimeException](#) をしてください

のスロー

のは、をスローするをしています。

```
public void checkNumber(int number) throws IllegalArgumentException {
    if (number < 0) {
        throw new IllegalArgumentException("Number must be positive: " + number);
    }
}
```

は3にスローされます。このステートメントは2つのにけることができます

- `new IllegalArgumentException(...)`は、`IllegalArgumentException`クラスのインスタンスをしており、がしているエラーをするメッセージがされます。
- `throw ...`オブジェクトをスローします。

がスローされると、がされるまで、むステートメントがします。これはのでされています。

のように、オブジェクトをしてのステートメントにスローすることをおめします。また、プログラマーにのをさせるために、のあるエラーメッセージをにめることをおめします。ただし、ずしもエンドユーザーにするがあるメッセージではありません。めに、Javaはメッセージのをサポートしていません。

いくつかのをするがあります

- `checkNumber`をthrows `IllegalArgumentException`としてしました。`IllegalArgumentException`はチェックされたであるため、これはにはではありません。「[Javaの - チェックとチェック](#)」をしてください。ただし、これをうこと、またメソッドのjavadocコメントにスローされたをめることはいです。
- `throw`ののコードにできません。したがって、もしたちがこれをいたら

```
throw new IllegalArgumentException("it is bad");
return;
```

コンパイラは`return`のコンパイルエラーをします。

チェーニング

くのは、の`message`にえて2の`cause`をつコンストラクタがあります。`cause`により、をさ`cause`ことが出来ます。ここにがあります。

に、なエラーがしたときにアプリケーションがスローするチェックのをします。`cause`をけれるコンストラクタがまれていることにしてください。

```
public class AppErrorException extends RuntimeException {
    public AppErrorException() {
        super();
    }

    public AppErrorException(String message) {
        super(message);
    }

    public AppErrorException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

に、チェーンをすコードをします。

```

public String readFirstLine(String file) throws AppErrorException {
    try (Reader r = new BufferedReader(new FileReader(file))) {
        String line = r.readLine();
        if (line != null) {
            return line;
        } else {
            throw new AppErrorException("File is empty: " + file);
        }
    } catch (IOException ex) {
        throw new AppErrorException("Cannot read file: " + file, ex);
    }
}

```

try ブロックの throw はをし、なメッセージでをします。に、 catch ブロックの throw はしいチェックされたでそれをラップすることによって IOException しています。しかし、のをてているわけではありません。IOException を cause としてすることで、スタックトレースの **とみみ**でしたように、スタックトレースにできるように IOException をします。

カスタム

ほとんどのでは、Exception をスローするときの Exception クラスをするがコードのからはです。これは、なエラーメッセージをえるためにがなににてはまります。その、チェックされたではないため、 **RuntimeException** がはされます。なクラスのエラーにしては、のクラスがします。

- **UnsupportedOperationException** - のがサポートされていない
- **IllegalArgumentException** - なパラメータがメソッドにされた
- **IllegalStateException** - あなたのAPIにはしてこらないになっているか、APIをなでしたします

カスタムクラスをいたいんケースにはのがあります。

- のがするためのAPIまたはライブラリをしており、APIのユーザーがAPIのをにしてできるようにしたいは、のをよりなとすることができます。
- プログラムのあるでのエラーのをスローしています。これは、プログラムののでしてするもので、のよりなエラーとするがあります。

チェックされていないにして RuntimeException をするか、RuntimeException のサブクラスでない Exception をして Exception をチェックすることで、のカスタムをすることができます。

RuntimeException のサブクラスではない Exception のサブクラスは、をチェックします。

```

public class StringTooLongException extends RuntimeException {
    // Exceptions can have methods and fields like other classes
    // those can be useful to communicate information to pieces of code catching
    // such an exception
    public final String value;
    public final int maximumLength;

    public StringTooLongException(String value, int maximumLength){

```

```

        super(String.format("String exceeds maximum Length of %s: %s", maxLength, value));
        this.value = value;
        this.maxLength = maxLength;
    }
}

```

それらはあらかじめされたとしてすることができます

```

void validateString(String value){
    if (value.length() > 30){
        throw new StringTooLongException(value, 30);
    }
}

```

また、がキャッチされてされるでフィールドをできます。

```

void anotherMethod(String value){
    try {
        validateString(value);
    } catch (StringTooLongException e){
        System.out.println("The string '" + e.value +
            "' was longer than the max of " + e.maxLength );
    }
}

```

オラクルの[Javaドキュメンテーション](#)によれば、のことにしてください。

[...]クライアントがからリカバリすることがにされるは、チェックにしてください。クライアントがからするためにもできないは、チェックされていないにしてください。

もっと

- [なぜRuntimeExceptionはなをとしないのですか](#)

try-with-resources ステートメント

Java SE 7

[try-catch-final ステートメント](#)のがすように、 `finally` をするリソースクリーンアップでは、エッジケースをしくするためにかなりのの "ボイラープレート" コードがです。 Java 7は、このを *try-with-resources* ステートメントのでうもっとなをします。

リソースとはですか

Java 7は `java.lang.AutoCloseable` インタフェースをし、 *try-with-resources* ステートメントをしてクラスをできるようにしました。 `AutoCloseable` をするクラスのインスタンスは、 リソースとされます。これらは、ガベージコレクタをするのではなく、にするがあります。

`AutoCloseable` インターフェイスは、のメソッドをします。

```
public void close() throws Exception
```

`close()` メソッドはリソースをなでするがあります。このでは、すでにされたリソースでメソッドをびすことがであるべきとべています。さらに、`AutoCloseable` をするクラスは、`Exception` よりもなをスローするように `close()` メソッドをすることをくします。

いJavaクラスとインタフェースが `AutoCloseable` してい `AutoCloseable`。これらには、

- `InputStream`、`OutputStream` およびそのサブクラス
- `Reader`、`Writer`、およびそれらのサブクラス
- `Socket` と `ServerSocket` とそのサブクラス
- `Channel` とそのサブクラス、および
- JDBCインタフェース `Connection`、`Statement`、`ResultSet`、およびそれらのサブクラス

アプリケーションとのクラスもこれをうがあります。

な **try-with-resource** ステートメント

try-with-resources のは、な *try-catch*、*try-finally* および *try-catch-finally* フォームについています。ここには「な」のがあります。 *catch* ないフォーム、または *finally*。

```
try (PrintStream stream = new PrintStream("hello.txt")) {  
    stream.println("Hello world!");  
}
```

するリソースは、`try` のの (...) セクションでとしてされます。のでは、リソース `stream` をし、それをしくされた `PrintStream` します。

リソースがされると、`try` ブロックがされます。これがすると、`stream.close()` がにびされ、リソースがれないことがされます。ブロックがどのようにしても、`close()` コールがすることにしてください。

された **try-with-resource** ステートメント

pre-Java 7 の *try-catch-finally* のように、*try-with-resources* ステートメントを *catch* および *finally* ブロックですることができます。のコードスニペットは、`PrintStream` コンストラクタがスローできる `FileNotFoundException` をするために、のものに *catch* ブロックをします。

```
try (PrintStream stream = new PrintStream("hello.txt")) {  
    stream.println("Hello world!");  
} catch (FileNotFoundException ex) {  
    System.err.println("Cannot open the file");  
} finally {  
    System.err.println("All done");  
}
```


リソースのまたはtryブロックがをスローすると、catchブロックがされます。finallyブロックは、のtry-catch-finallyステートメントとに、にされます。

しかし、すべきことがいくつかあります

- リソースはcatchおよびfinallyブロックのスコープです。
- ステートメントがcatchブロックにするに、リソースのクリーンアップがわれます。
- リソースクリーンアップがをスローした、それはcatchブロックの1つにcatchれるがあります。

のリソースの

のコードスニペットは、されているのリソースをしています。、リソースのは、1つのステートメントでのリソースをできます。えは

```
try (InputStream is = new FileInputStream(file1);
     OutputStream os = new FileOutputStream(file2)) {
    // Copy 'is' to 'os'
}
```

これはりにします。tryブロックのでは、isとosisにじられます。すべきがいくつかあります。

- はコードにわれ、のリソースはのもののをすることができます。
- にされたリソースはすべてクリーンアップされます。
- リソースは、のでクリーンアップされます。

したがって、のでは、isにされているosとしたにクリーンアップ、およびisにがあるにクリーンアップされるos。

try-with-resourceとなtry-catch-finallyの

Javaでは、try-catch-finallyステートメントのからtry-with-resourceフォームのをしています。はJLSをしてください

たとえば、このなtry-with-resourceはのとおりです。

```
try (PrintStream stream = new PrintStream("hello.txt")) {
    stream.println("Hello world!");
}
```

try-catch-finallyとにされています

```
// Note that the constructor is not part of the try-catch statement
PrintStream stream = new PrintStream("hello.txt");

// This variable is used to keep track of the primary exception thrown
// in the try statement. If an exception is thrown in the try block,
```

```
// any exception thrown by AutoCloseable.close() will be suppressed.
Throwable primaryException = null;

// The actual try block
try {
    stream.println("Hello world!");
} catch (Throwable t) {
    // If an exception is thrown, remember it for the finally block
    primaryException = t;
    throw t;
} finally {
    if (primaryException == null) {
        // If no exception was thrown so far, exceptions thrown in close() will
        // not be caught and therefore be passed on to the enclosing code.
        stream.close();
    } else {
        // If an exception has already been thrown, any exception thrown in
        // close() will be suppressed as it is likely to be related to the
        // previous exception. The suppressed exception can be retrieved
        // using primaryException.getSuppressed().
        try {
            stream.close();
        } catch (Throwable suppressedException) {
            primaryException.addSuppressed(suppressedException);
        }
    }
}
}
```

JLSでは、`primaryException`はのJavaコードにはえないとしています。

*try-with-resources*のは、とのものとしてされています。えは

```
try (PrintStream stream = new PrintStream(fileName)) {
    stream.println("Hello world!");
} catch (NullPointerException ex) {
    System.err.println("Null filename");
} finally {
    System.err.println("All done");
}
```

のものとです。

```
try {
    try (PrintStream stream = new PrintStream(fileName)) {
        stream.println("Hello world!");
    }
} catch (NullPointerException ex) {
    System.err.println("Null filename");
} finally {
    System.err.println("All done");
}
```

スタックトレースのとみみ

オブジェクトがされるとつまり、`new` オブジェクトをするとき、`Throwable` コンストラクタはがされたコンテキストにするをします。その、このをスタックトレースのすることができます。ス

スタックトレースは、にのとなったのにちます。

スタックトレースの

スタックトレースのは`printStackTrace()` メソッドをびすだけです。えば

```
try {
    int a = 0;
    int b = 0;
    int c = a / b;
} catch (ArithmeticException ex) {
    // This prints the stacktrace to standard output
    ex.printStackTrace();
}
```

のない`printStackTrace()` メソッドは、アプリケーションのにします。の`System.out` です。

`printStackTrace(PrintStream)` と `printStackTrace(PrintWriter)` も、された`Stream` または `Writer` されます。

ノート

1. `stacktrace` には、のはまれません。これらのをするには、`toString()` メソッドをできます。えは

```
// Print exception and stacktrace
System.out.println(ex);
ex.printStackTrace();
```

2. `Stacktrace` はえめにするがあります。 [とし - またはなスタックトレースを](#) してください。ロギングフレームワークをし、ログにするオブジェクトをすがいがあります。

スタックトレースの

2つのファイルで2つのクラスからなるのなプログラムをえてみましょう。 のためにファイルとをしました。

```
File: "Main.java"
1  public class Main {
2      public static void main(String[] args) {
3          new Test().foo();
4      }
5  }

File: "Test.java"
1  class Test {
2      public void foo() {
3          bar();
4      }
5
6      public int bar() {
```

```

7         int a = 1;
8         int b = 0;
9         return a / b;
10    }

```

これらのファイルをコンパイルしてすると、のようになられます。

```

Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Test.bar(Test.java:9)
    at Test.foo(Test.java:3)
    at Main.main(Main.java:3)

```

たちにをえているのかをするために、このをにんでみましょう。

1は、キャッチされないのために「main」というスレッドがしたことをしています。のなは `java.lang.ArithmeticException` で、メッセージは `"/ by zero"` です。

このjavadocをべると、のようになります。

ながしたときにスローされます。たとえば、「ゼロでる」は、このクラスのインスタンスをスローします。

、`"/ by zero"`というメッセージは、のが、らかのコードがかを0でりろうとしているといういです。しかし

りの3はスタックトレースです。は、コールスタックのメソッドまたはコンストラクタコールをし、それぞれがの3つのことをします。

- されていたクラスとメソッドの、
- ソースコードファイル、
- されていたステートメントのソースコード

これらのスタックトレースのには、のコールのフレームがにされます。のののフレームは `Test.bar` メソッドにあり、`Test.java` ファイルの9にあります。それはのです

```
return a / b;
```

々はどこにのファイルののカップルをてみると、ことがらかである、ゼロのをつことになり。これがのであることはいありません。

さらにめるがあるは、`stacktrace`から `bar()` が `Test.java` の3の `foo()` からびされ、 `foo()` が `Main.main()` からびされたことが `Main.main()` ます。

スタックフレームのクラスとメソッドのは、クラスとメソッドのです。あなたはのようないケースをするがあります

- ねクラスまたはクラスは、`"OuterClass $ InnerClass"`のようになります。
- のクラスは、`"OuterClass $ 1"`、`"OuterClass $ 2"`などのようになります。

- ・ コンストラクタのコード、インスタンスフィールドイニシャライザまたはインスタンスイニシャライザブロックがされている、メソッドは "" になります。
- ・ フィールドイニシャライザまたはイニシャライザブロックのコードがされている、メソッドは "" になります。

Javaのいくつかのバージョンでは、なのためにアプリケーションがしたときにするがあるように、スタックトレースコードはりしスタックフレームシーケンスをしてします。

チェーンとれスタックトレース

Java SE 1.4

のは、あるコードがをキャッチした、しいをしてスローし、のをとしてしたときにします。にをします。

```
File: Test.java
1  public class Test {
2      int foo() {
3          return 0 / 0;
4      }
5
6      public Test() {
7          try {
8              foo();
9          } catch (ArithmeticException ex) {
10             throw new RuntimeException("A bad thing happened", ex);
11          }
12      }
13
14      public static void main(String[] args) {
15          new Test();
16      }
17 }
```

のクラスをコンパイルしてすると、のようなスタックトレースがされます。

```
Exception in thread "main" java.lang.RuntimeException: A bad thing happened
    at Test.<init>(Test.java:10)
    at Test.main(Test.java:15)
Caused by: java.lang.ArithmeticException: / by zero
    at Test.foo(Test.java:3)
    at Test.<init>(Test.java:8)
    ... 1 more
```

stacktraceは、クラス、メソッド、およびコールスタックでまり、これはアプリケーションがクラッシュするがします。これには、causeをする「」がきます。クラスとメッセージがされ、いてのスタックフレームがされます。トレースは、のNのフレームがのとじであることをす「... N more」でします。

""は、プライマリのcauseがnullでないにのみにまれnull。はにできます。その、スタックトレースはの「」トレースをつことができます。

causeメカニズムはJava 1.4.0のThrowable APIでのみされていました。これによって、をすためにカスタムフィールドをするアプリケーションと、カスタムのprintStackTraceメソッドをしてチェーンをするがありました。

スタックトレースをとしてみ

によっては、アプリケーションがJava Stringとしてスタックトレースをできるがあるため、のにできます。これをうなアプローチは、なOutputStreamまたはWriterをしてメモリのバッファにきみ、それをprintStackTrace(...)すことprintStackTrace(...)。

Apache CommonsとGuavaライブラリは、スタックトレースをとにするユーティリティメソッドをしています。

```
org.apache.commons.lang.exception.ExceptionUtils.getStackTrace(Throwable)

com.google.common.base.Throwables.getStackTraceAsString(Throwable)
```

コードベースでサードパーティライブラリをできないは、のでタスクをします。

```
/**
 * Returns the string representation of the stack trace.
 *
 * @param throwable the throwable
 * @return the string.
 */
public static String stackTraceToString(Throwable throwable) {
    StringWriter stringWriter = new StringWriter();
    throwable.printStackTrace(new PrintWriter(stringWriter));
    return stringWriter.toString();
}
```

スタックトレースをすることをしているは、getStackTrace()およびgetCause()をするが、スタックトレースをするよりもです。

InterruptedExceptionの

InterruptedExceptionはするですThread.sleep()のようなメソッドにThread.sleep()ますが、ってすると、ではあまりしないコードになります。

InterruptedExceptionがされたは、コードがのスレッドでThread.interrupt()をThread.interrupt()ことをします。「これはのコードですはしてしません"したがって、のようなことをしてください

```
// Bad. Don't do this.
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    // disregard
}
```

しかし、これはまさに「な」をするったです。アプリケーションで`InterruptedException`がすることがないことがかっているは、そのようなイベントをプログラムのになとしてい、できるだけするがあります。

「な」りみをするなは、のようなものです。

```
// When nothing will interrupt your code
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    throw new AssertionError(e);
}
```

これは2つのことをいいます。 `InterruptedException`がにスローされていないかのように、スレッドのりみステータスがにされ、アプリケーションのながられたことをす `AssertionError`スローされます。 `catch` ブロックがしてしてはならないので、このコードがするスレッドをしないことがであることがかっているは、このコードはです。

Guavaの`Uninterruptibles`クラスをすると、このパターンがされます。

`Uninterruptibles.sleepUninterruptibly()`びすと、スリープがするまでスレッドのがされますので、での`InterruptedException`をしてスローするためにびされます。あなたがこのようなコードをしてしないことをっていれば、これはtry-catchブロックであなたのスリープコールをラップするをします。

ただし、スレッドがされないことをすることはできません。に、 `Executor`やのスレッドによってされるコードをしている、コードがりみににすることがです。そうしないと、アプリケーションがストールしたり、デッドロックしたりします。

このようなには、 `InterruptedException`がコールスタックをできるようにするのがもいです。つまり、 `throws InterruptedException`をメソッドにに`throws InterruptedException`します。これはクルーシングにえるかもしれませんが、にはましいプロパティです。メソッドのシグネチャは、びしにりみににすることをします。

```
// Let the caller determine how to handle the interrupt if you're unsure
public void myLongRunningMethod() throws InterruptedException {
    ...
}
```

られたたとえば、チェックされたを`throw`しないメソッドをオーバーライドしているは、りみをするためににされるコードをして、をさせずにりみステータスをリセットできます。これはをするのをさせますが、にするわけではありません。

```
// Suppresses the exception but resets the interrupted state letting later code
// detect the interrupt and handle it properly.
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
}
```

```
return ...; // your expectations are still broken at this point - try not to do more work.
}
```

Java - チェックされていないとチェックされた

すべてのJavaは、Exceptionクラスのクラスのインスタンスです。これはのようすことができます。

- `java.lang.Throwable` - これはすべてのクラスのクラスです。そのメソッドとコンストラクタは、すべてののにするのをします。
 - `java.lang.Exception` - これは、すべてのののスーパークラスです。
 - さまざまなクラスとカスタムクラス。
 - `java.lang.RuntimeException` - チェックのであるすべてのののスーパークラス。
 - さまざまなおよびカスタムランタイムクラス。
 - `java.lang.Error` - これはすべての "なエラー" のスーパークラスです。

ノート

1. チェックし、チェックのはのりです。
2. `Throwable`、`Exception`および`RuntimeException`クラスは、`abstract`クラスとしてうがあります。
 - [Pitfall - Throwable Throwable、Exception、ErrorまたはRuntimeExceptionをしてください](#)。
3. `Error`は、アプリケーションがしようとするのがでない、またはでないでJVMによってスローされます。
4. `Throwable`カスタムサブタイプをすることはではありません。Javaツールとライブラリでは、`Error`と`Exception`が`Throwable`のサブタイプであるとし、そのがっているとするがあります。

チェックされたとチェックの

いくつかのプログラミングにおけるサポートのの1つは、えられたメソッドまたはプロシージャがスローするのあるをることがしいということです。されないがプログラムをクラッシュさせるがあることをえると、がのになることがあります。

Javaは、このをチェックされたメカニズムにします。まず、Javaはを2つのカテゴリにします。

- チェックされたは、アプリケーションができなければならないされるイベントをします。たとえば、`IOException`およびそのサブタイプは、でするのあるエラーをします。たとえば、ファイルやディレクトリがしないためにファイルがくことができず、ネットワークのがされたためにネットワークのみりときみにするなどのがあります。
- チェックされていないは、アプリケーションができないしないイベントをします。これらは、アプリケーションのバグのです。

では、「スローされた」とは、`throw`によってに`throw`された、に、キャストなどでスローされたをしますに、「」

とは、ネストされたびしであり、そのびしのでされません。のサンプルコードでこれをします。

チェックされたメカニズムの2のは、チェックされたがするのあるメソッドにがあることです。

- チェックは、メソッドでスローまたはされるは、いずれかのでキャッチしなければならない、またはメソッドのは、にされている`throws`を。 [このでは](#)、`throws`のがされています。
- チェックされたがイニシャライザブロックにスローされたりされた、そのブロックをキャッチしなければなりません。
- チェックされたは、フィールドのメソッドびしではできません。 そのようなをキャッチするはありません。

するに、チェックはされるか、されなければなりません。

これらののは、チェックされていないにはされません。これには、すべてのケースでチェックのがスローされるため、にがスローされるすべてのケースがまれます。

チェックされたの

これらのコードスニペットは、チェックされたのをするためのものです。いずれのもの、コンパイルエラーのあるバージョンのコードと、エラーがされた2のバージョンがされます。

```
// This declares a custom checked exception.
public class MyException extends Exception {
    // constructors omitted.
}

// This declares a custom unchecked exception.
public class MyException2 extends RuntimeException {
    // constructors omitted.
}
```

のは、にスローされたチェックされたがメソッドでされるべきでないに "げられる"とされるをしています。

```
// INCORRECT
public void methodThrowingCheckedException(boolean flag) {
    int i = 1 / 0; // Compiles OK, throws ArithmeticException
    if (flag) {
        throw new MyException(); // Compilation error
    } else {
        throw new MyException2(); // Compiles OK
    }
}

// CORRECTED
public void methodThrowingCheckedException(boolean flag) throws MyException {
    int i = 1 / 0; // Compiles OK, throws ArithmeticException
    if (flag) {
        throw new MyException(); // Compilation error
    } else {
        throw new MyException2(); // Compiles OK
    }
}
```

```
}
```

2のは、されたチェックをどのようにできるかをしています。

```
// INCORRECT
public void methodWithPropagatedCheckedException() {
    InputStream is = new FileInputStream("someFile.txt"); // Compilation error
    // FileInputStream throws IOException or a subclass if the file cannot
    // be opened.  IOException is a checked exception.
    ...
}

// CORRECTED (Version A)
public void methodWithPropagatedCheckedException() throws IOException {
    InputStream is = new FileInputStream("someFile.txt");
    ...
}

// CORRECTED (Version B)
public void methodWithPropagatedCheckedException() {
    try {
        InputStream is = new FileInputStream("someFile.txt");
        ...
    } catch (IOException ex) {
        System.out.println("Cannot open file: " + ex.getMessage());
    }
}
```

のは、フィールドイニシャライザでチェックをするをしています。

```
// INCORRECT
public class Test {
    private static final InputStream is =
        new FileInputStream("someFile.txt"); // Compilation error
}

// CORRECTED
public class Test {
    private static final InputStream is;
    static {
        InputStream tmp = null;
        try {
            tmp = new FileInputStream("someFile.txt");
        } catch (IOException ex) {
            System.out.println("Cannot open file: " + ex.getMessage());
        }
        is = tmp;
    }
}
```

このケースでは、にりてること`is`できないをしなけりばならず、のもりてられるがあることにしてください。

き

は、プログラムのにするエラーです。の2つのをするJavaプログラムをえてみましょう。

```

class Division {
    public static void main(String[] args) {

        int a, b, result;

        Scanner input = new Scanner(System.in);
        System.out.println("Input two integers");

        a = input.nextInt();
        b = input.nextInt();

        result = a / b;

        System.out.println("Result = " + result);
    }
}

```

はこのコードをコンパイルしてし、0でしようとするをてみましょう

```

Input two integers
7 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Division.main(Division.java:14)

```

0によるは、としてできないをするなです。Javaではをスローすることでこれをします。この、は`ArithmeticException`クラスのインスタンスです。

スタックトレースのとみりのでは、2つののののをしています。

のは、それがすフローです。をせずに、こののなは、に `b == 0`

```

class Division {
    public static void main(String[] args) {

        int a, b, result;

        Scanner input = new Scanner(System.in);
        System.out.println("Input two integers");

        a = input.nextInt();
        b = input.nextInt();

        if (b == 0) {
            System.out.println("You cannot divide by zero.");
            return;
        }

        result = a / b;

        System.out.println("Result = " + result);
    }
}

```

これは `You cannot divide by zero.` メッセージを `You cannot divide by zero.` ユーザーがゼロでしよう
とすると、コンソールをし、プログラムをなでします。をしてこのをするのは、`if` フローを `try-`

catch ブロックでできること if。

```
...

a = input.nextInt();
b = input.nextInt();

try {
    result = a / b;
}
catch (ArithmeticException e) {
    System.out.println("An ArithmeticException occurred. Perhaps you tried to divide by
zero.");
    return;
}

...
```

try catch ブロックはのようになされます。

1. try ブロックでコードのをします。
2. try ブロックでがしたは、すぐにし、このがでキャッチされたかどうかをする catch ブロックのインスタンスであるとき、このケースで ArithmeticException。
3. がキャッチされた、それは e いてられ、 catch ブロックがされます。
4. try または catch ブロックのいずれかがしたつまり、コードにキャッチされないがしない、きき try-catch ブロックののコードをします。

、アプリケーションののフローのとしてをし、がまたはしないものになるようにすることをおめします。たとえば、メソッドがしたときに null をすわりに、をスローして、そのメソッドをするアプリケーションが、にしたのをしてにするのフローをできるようにするがいでしょう。あるでは、のをさなければならぬというをしします。したのをすためにののがスローされるがあります。

をしないとのについては、「[Javaのとし - の](#)」をしてください。

try catch ブロックのをす

いですが、ブロックにの return をすることはです

```
public static int returnTest(int number){
    try{
        if(number%2 == 0) throw new Exception("Exception thrown");
        else return x;
    }
    catch(Exception e){
        return 3;
    }
    finally{
        return 7;
    }
}
```

このメソッドは、try / catchブロックにけられたfinallyブロックがかがされるにされるため、に7をします。さて、にreturn 7;これはtry / catchのりによってわるものです。

キャッチブロックがプリミティブをし、そのプリミティブがfinallyブロックでされた、キャッチブロックでされたがされ、finallyブロックからののはされます。

のでは、「1」ではなく「0」がされます。

```
public class FinallyExample {

    public static void main(String[] args) {
        int n = returnTest(4);

        System.out.println(n);
    }

    public static int returnTest(int number) {

        int returnNumber = 0;

        try {
            if (number % 2 == 0)
                throw new Exception("Exception thrown");
            else
                return returnNumber;
        } catch (Exception e) {
            return returnNumber;
        } finally {
            returnNumber = 1;
        }
    }
}
```

のな

このでは、のいくつかのなについてします。

プログラムでびしスタックをべる

Java SE 1.4

スタックトレースのなは、アプリケーションエラーとそのコンテキストにするをし、プログラマがをしてできるようにすることです。々のもののためにすることができます。たとえば、SecurityManagerクラスは、びしをとっているコードをするがあるかどうかをするためにびしスタックをべるがあります。

のようにをしてプログラムスタックでびしスタックをべることができます。

```
Exception ex = new Exception();    // this captures the call stack
StackTraceElement[] frames = ex.getStackTrace();
System.out.println("This method is " + frames[0].getMethodName());
System.out.println("Called from method " + frames[1].getMethodName());
```

これにはいくつかのながあります

1. `StackTraceElement` ではありません。 `printStackTrace` でされるよりくのはありません。 フレームのローカルのはできません。
2. `getStackTrace()` は、JVMがフレームをすことをされていることをしています。

のマシンでは、によってはスタックトレースから1つのスタックフレームをすることがあります。な、このスローオブジェクトにするスタックトレースをたないマシンは、このメソッドからゼロをすことができます。

の

のところでべたように、のは、のスレッドのすべてのスタックフレームにするをキャプチャしてするため、かなりです。によっては、そのがのにしてされることはないことがわかっています。例えば、スタックトレースはしてされません。その、をキャプチャしないようにカスタムでできるのみがあります。

スタックトレースになスタックフレームは、 `Throwable` コンストラクタが

`Throwable.fillInStackTrace()` メソッドをびすときにされます。このメソッドは `public` です。つまり、サブクラスはそれをオーバーライドできます。このトリックは、 `Throwable` からしたメソッドをもしないメソッドでオーバーライドすることです。例えば

```
public class MyException extends Exception {
    // constructors

    @Override
    public void fillInStackTrace() {
        // do nothing
    }
}
```

このアプローチのは、 `fillInStackTrace()` をオーバーライドするがスタックトレースをできないことと、なシナリオではにたないことです。

スタックトレースのまたはきえ

Java SE 1.4

によっては、のでされたのスタックトレースに、ったまたはがユーザーにしたいくないがまれていることがあります。これらのシナリオでは、 `Throwable.setStackTrace` をして、をす `StackTraceElement` オブジェクトのをきえることができます。

たとえば、のようにしてのスタックをできます。

```
exception.setStackTrace(new StackTraceElement[0]);
```

された

Java SE 7

Java 7では、*try-with-resources*と、それにするのがされました。のスニペットをえてみましょう。

```
try (Writer w = new BufferedWriter(new FileWriter(someFilename))) {
    // do stuff
    int temp = 0 / 0;    // throws an ArithmeticException
}
```

がスローされると、`try`は`w`で`close()`をびし、バッファされたをフラッシュし、`FileWriter`ます。しかし、をフラッシュするに`IOException`がスローされるとどうなりますか

がこるかは、リソースをクリーンアップしているにスローされたはすべてされることです。がキャッチされ、プライマリのされたリストにされます。に、*try-with-resources*はのリソースのクリーンアップをします。に、がスローされます。

のパターンは、リソースのにがスローされた、または`try`ブロックがにしたにします。スローされたのがなになり、そののクリーンアップでするはされます。

されたは、`getSuppressedExceptions`をびすことによって、プライマリオブジェクトからできます。

try-finallyおよび**try-catch-finally**ステートメント

`try...catch...finally`は、とクリーンアップコードをみわせています。`finally`ブロックには、すべてのでされるコードがまれています。これにより、リソースやののクリーンアップにしています。

Try-finally

はな `try...finally` フォームのです

```
try {
    doSomething();
} finally {
    cleanUp();
}
```

`try...finally`のはのようになります。

- `try` ブロックのコードがされます。
- `try` ブロックにがスローされなかった
 - `finally` ブロックのコードがされます。

- `finally` ブロックがスローすると、そのがします。
- それのは、 `try...finally` にのにか `try...finally` 。
- `try` ブロックにかスローされた
 - `finally` ブロックのコードがされます。
 - `finally` ブロックがスローすると、そのがします。
 - それの、のはききしけます。

`finally` ブロックのコードはにされます。のは、 `System.exit(int)` がびされた、またはJVMがパニックするです。したがって、 `finally` ブロックはにするがあるしいプレースコードです。ファイルやそののリソースをじる、ロックをするなど。

try-catch-finally

2のは `catch` と `finally` がどのようににできるかをしています。また、リソースのクリーンアップがではないこともしています。

```
// This code snippet writes the first line of a file to a string
String result = null;
Reader reader = null;
try {
    reader = new BufferedReader(new FileReader(fileName));
    result = reader.readLine();
} catch (IOException ex) {
    Logger.getLogger().warn("Unexpected IO error", ex); // logging the exception
} finally {
    if (reader != null) {
        try {
            reader.close();
        } catch (IOException ex) {
            // ignore / discard this exception
        }
    }
}
```

こので `try...catch...finally` ななのなセットはここではするのがしいです。なバージョンは、 `finally` ブロックのコードがにされるとということです。

これをリソースのからると

- たちは `try` ブロックのに "リソース"つまり `reader` をして、 `finally` ブロックのスコープにるよう `finally` ます。
- `new FileReader(...)` をくことで、 `catch` はファイルをくときにスローされた `IOException` をできます。
- `finally` ブロックには `reader.close()` がです。これは、 `try` ブロックまたは `catch` ブロックでできないパスがいくつかするためです。
- ただし、 `reader` がされるにかスローされているがあるため、な `null` もです。
- に、 `reader.close()` コールはにをスローするがあります。これについてはにしますが、ソースでをキャッチしないと、コールスタックをさらにするがあります。

Java 7では、[resource-try-with-resources](#)をしてリソースのクリーンアップをにしています。

メソッドの 'throws'

Javaのチェックメカニズムでは、のメソッドがされたチェックをスローできることをプログラマがするがあります。これは`throws`をしてわれます。例えば

```
public class OddNumberException extends Exception { // a checked exception
}

public void checkEven(int number) throws OddNumberException {
    if (number % 2 != 0) {
        throw new OddNumberException();
    }
}
```

`throws OddNumberException`へのびしことをし`checkEven`のあるスローする `OddNumberException`。

`throws`はのリストをすることができ、チェックされていないとされたをめることができます。

```
public void checkEven(Double number)
    throws OddNumberException, ArithmeticException {
    if (!Double.isFinite(number)) {
        throw new ArithmeticException("INF or NaN");
    } else if (number % 2 != 0) {
        throw new OddNumberException();
    }
}
```

チェックされていないをスローするとするのはですか

メソッドの`throws`には2つのがあります。

1. これはコンパイラにがスローされたことをし、コンパイラがキャッチされていないチェックされたをエラーとしてできるようにします。
2. これは、メソッドをびすコードをしているプログラマに、のをします。こののために、チェックされていないを`throws`リストにめることがよくあります。

`throws`リストは、APIドキュメントのにはjavadocツールでされ、なIDEの「ホバーテキスト」メソッドのヒントでもされます。

スローとメソッドのオーバーライド

`throws`は、メソッドオーバーライドのでメソッドのシグネチャのをします。オーバーライドメソッドは、オーバーライドされたメソッドによってスローされるのと同じチェックセット、またはサブセットでできます。ただし、オーバーライドメソッドはのチェックをすることはできません。例えば

```

@Override
public void checkEven(int number) throws NullPointerException // OK-NullPointerException is an
unchecked exception
    ...

@Override
public void checkEven(Double number) throws OddNumberException // OK-identical to the
superclass
    ...

class PrimeNumberException extends OddNumberException {}
class NonEvenNumberException extends OddNumberException {}

@Override
public void checkEven(int number) throws PrimeNumberException, NonEvenNumberException //
OK-these are both subclasses

@Override
public void checkEven(Double number) throws IOException // ERROR

```

このルールのは、オーバーライドされたメソッドがオーバーライドされたメソッドがスローできなかったチェックをスローできる、そのメソッドはのをするためです。

オンラインでとをむ <https://riptutorial.com/ja/java/topic/89/>と

Examples

SecurityManagerの

JavaマシンJVMは、SecurityManagerがインストールされたでできます。 SecurityManagerは、コードがロードされたやコードにするためにされたなどのについて、JVMでされているコードのをします。

SecurityManagerは、JVMのにコマンドラインでjava.security.managerシステムプロパティをすることでインストールできます。

```
java -Djava.security.manager <main class name>
```

Javaコードからプログラムに

```
System.setSecurityManager(new SecurityManager())
```

のJava SecurityManagerは、ポリシーファイルにされているポリシーについてアクセスをします。ポリシーファイルがされていないは、\$JAVA_HOME/lib/security/java.policyにあるデフォルトのポリシーファイルがされます。

ClassLoaderによってロードされたサンドボックスクラス

ClassLoaderは、コードのソースをするProtectionDomainをするがあります。

```
public class PluginClassLoader extends ClassLoader {
    private final ClassProvider provider;

    private final ProtectionDomain pd;

    public PluginClassLoader(ClassProvider provider) {
        this.provider = provider;
        Permissions permissions = new Permissions();

        this.pd = new ProtectionDomain(provider.getCodeSource(), permissions, this, null);
    }

    @Override
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        byte[] classDef = provider.getClass(name);
        Class<?> clazz = defineClass(name, classDef, 0, classDef.length, pd);
        return clazz;
    }
}
```

loadClassではなく findClassオーバーライドすることによって、モデルはされ、PluginClassLoader

はまずクラスのためにシステムとクラスローダーにいさせます。

ポリシーをする

```
public class PluginSecurityPolicy extends Policy {
    private final Permissions appPermissions = new Permissions();
    private final Permissions pluginPermissions = new Permissions();

    public PluginSecurityPolicy() {
        // amend this as appropriate
        appPermissions.add(new AllPermission());
        // add any permissions plugins should have to pluginPermissions
    }

    @Override
    public Provider getProvider() {
        return super.getProvider();
    }

    @Override
    public String getType() {
        return super.getType();
    }

    @Override
    public Parameters getParameters() {
        return super.getParameters();
    }

    @Override
    public PermissionCollection getPermissions(CodeSource codesource) {
        return new Permissions();
    }

    @Override
    public PermissionCollection getPermissions(ProtectionDomain domain) {
        return isPlugin(domain)?pluginPermissions:appPermissions;
    }

    private boolean isPlugin(ProtectionDomain pd){
        return pd.getClassLoader() instanceof PluginClassLoader;
    }
}
```

に、ポリシーとSecurityManagerをしますデフォルトのはです。

```
Policy.setPolicy(new PluginSecurityPolicy());
System.setSecurityManager(new SecurityManager());
```

ポリシールールの

のProtectionDomainにするのPermissionをすることは、ドメインがするのアクセスになく、することがましいことがあります。このは、こののをたすためのすべてのなアプローチの1つをしています。「ネガティブ」パーミッションクラスとラッパーをし、デフォルトPolicyをそのようなパーミッションのリポジトリとしてできるようにします。

ノ一ト

- に、ポリシーファイルのとメカニズムのりてはをけません。つまり、ポリシーファイルのルールはききグラントとしてされます。
- ポリシーラッパーは、デフォルトのファイルバックアップ`Policy`には`com.sun.security.provider.PolicyFile`をカプセルするためのものです。
- されたアクセスは、ポリシーレベルでのみされます。ドメインににりてられている、デフォルトではそのドメインによっての「な」アクセスとしてわれます。

DeniedPermission クラス

```
package com.example;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Modifier;
import java.security.BasicPermission;
import java.security.Permission;
import java.security.UnresolvedPermission;
import java.text.MessageFormat;

/**
 * A representation of a "negative" privilege.
 * <p>
 * A <code>DeniedPermission</code>, when "granted" (to some <code>ProtectionDomain</code>
and/or
 * <code>Principal</code>), represents a privilege which <em>cannot</em> be exercised,
regardless of
 * any positive permissions (<code>AllPermission</code> included) possessed. In other words,
if a
 * set of granted permissions, <em>P</em>, contains a permission of this class, <em>D</em>,
then the
 * set of effectively granted permissions is<br/>
 * <br/>
 * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;<em>{ P<sub>implied</sub> - D<sub>implied</sub> }</em>.
 * </p>
 * <p>
 * Each instance of this class encapsulates a <em>target permission</em>, representing the
 * "positive" permission being denied.
 * </p>
 * Denied permissions employ the following naming scheme:<br/>
 * <br/>
 *
 * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;<em>&lt;target_class_name>;:&lt;target_name>;(:&lt;target_actions>)</em>
 *
 * <br/>
 * where:
 * <ul>
 * <li><em>target_class_name</em> is the name of the target permission's class,</li>
 * <li><em>target_name</em> is the name of the target permission, and</li>
 * <li><em>target_actions</em> is, optionally, the actions string of the target
permission.</li>
 * </ul>
 *
 * A denied permission, having a target permission <em>t</em>, is said to <em>imply</em>
another
 * permission <em>p</em>, if:
 * <ul>
```

```

* <li>p <em>is not</em> itself a denied permission, and <code>(t.implies(p) == true)</code>,
* or</li>
* <li>p <em>is</em> a denied permission, with a target <em>t1</em>, and
* <code>(t.implies(t1) == true)</code>.
* </ul>
* <p>
* It is the responsibility of the policy decision point (e.g., the <code>Policy</code>
provider) to
* take denied permission semantics into account when issuing authorization statements.
* </p>
*/
public final class DeniedPermission extends BasicPermission {

    private final Permission target;
    private static final long serialVersionUID = 473625163869800679L;

    /**
     * Instantiates a <code>DeniedPermission</code> that encapsulates a target permission of
the
     * indicated class, specified name and, optionally, actions.
     *
     * @throws IllegalArgumentException
     *         if:
     *         <ul>
     *         <li><code>targetClassName</code> is <code>>null</code>, the empty string,
does not
     *         refer to a concrete <code>Permission</code> descendant, or refers to
     *         <code>DeniedPermission.class</code> or
<code>UnresolvedPermission.class</code>.</li>
     *         <li><code>targetName</code> is <code>>null</code>.</li>
     *         <li><code>targetClassName</code> cannot be instantiated, and it's the
caller's fault;
     *         e.g., because <code>targetName</code> and/or <code>targetActions</code> do
not adhere
     *         to the naming constraints of the target class; or due to the target class
not
     *         exposing a <code>(String name)</code>, or <code>(String name, String
actions)</code>
     *         constructor, depending on whether <code>targetActions</code> is
<code>null</code> or
     *         not.</li>
     *         </ul>
     */
    public static DeniedPermission newDeniedPermission(String targetClassName, String
targetName,
        String targetActions) {
        if (targetClassName == null || targetClassName.trim().isEmpty() || targetName == null)
{
            throw new IllegalArgumentException(
                "Null or empty [targetClassName], or null [targetName] argument was
supplied.");
        }
        StringBuilder sb = new StringBuilder(targetClassName).append(":").append(targetName);
        if (targetName != null) {
            sb.append(":").append(targetName);
        }
        return new DeniedPermission(sb.toString());
    }

    /**
     * Instantiates a <code>DeniedPermission</code> that encapsulates a target permission of

```

```

the class,
    * name and, optionally, actions, collectively provided as the <code>name</code> argument.
    *
    * @throws IllegalArgumentException
    *         if:
    *         <ul>
    *         <li><code>name</code>'s target permission class name component is empty,
does not
    *         refer to a concrete <code>Permission</code> descendant, or refers to
    *         <code>DeniedPermission.class</code> or
<code>UnresolvedPermission.class</code>.</li>
    *         <li><code>name</code>'s target name component is <code>empty</code></li>
    *         <li>the target permission class cannot be instantiated, and it's the
caller's fault;
    *         e.g., because <code>name</code>'s target name and/or target actions
component(s) do
    *         not adhere to the naming constraints of the target class; or due to the
target class
    *         not exposing a <code>(String name)</code>, or
    *         <code>(String name, String actions)</code> constructor, depending on
whether the
    *         target actions component is empty or not.</li>
    *         </ul>
    */
    public DeniedPermission(String name) {
        super(name);
        String[] comps = name.split(":");
        if (comps.length < 2) {
            throw new IllegalArgumentException(MessageFormat.format("Malformed name [{0}]
argument.", name));
        }
        this.target = initTarget(comps[0], comps[1], ((comps.length < 3) ? null : comps[2]));
    }

    /**
     * Instantiates a <code>DeniedPermission</code> that encapsulates the given target
permission.
     *
     * @throws IllegalArgumentException
     *         if <code>target</code> is <code>null</code>, a
<code>DeniedPermission</code>, or an
     *         <code>UnresolvedPermission</code>.
     */
    public static DeniedPermission newDeniedPermission(Permission target) {
        if (target == null) {
            throw new IllegalArgumentException("Null [target] argument.");
        }
        if (target instanceof DeniedPermission || target instanceof UnresolvedPermission) {
            throw new IllegalArgumentException("[target] must not be a DeniedPermission or an
UnresolvedPermission.");
        }
        StringBuilder sb = new
StringBuilder(target.getClass().getName()).append(":").append(target.getName());
        String targetActions = target.getActions();
        if (targetActions != null) {
            sb.append(":").append(targetActions);
        }
        return new DeniedPermission(sb.toString(), target);
    }

    private DeniedPermission(String name, Permission target) {

```

```

        super(name);
        this.target = target;
    }

    private Permission initTarget(String targetClassName, String targetName, String
targetActions) {
        Class<?> targetClass;
        try {
            targetClass = Class.forName(targetClassName);
        }
        catch (ClassNotFoundException cnfe) {
            if (targetClassName.trim().isEmpty()) {
                targetClassName = "<empty>";
            }
            throw new IllegalArgumentException(
                MessageFormat.format("Target Permission class [{0}] not found.",
targetClassName));
        }
        if (!Permission.class.isAssignableFrom(targetClass) ||
Modifier.isAbstract(targetClass.getModifiers())) {
            throw new IllegalArgumentException(MessageFormat
                .format("Target Permission class [{0}] is not a (concrete) Permission.",
targetClassName));
        }
        if (targetClass == DeniedPermission.class || targetClass ==
UnresolvedPermission.class) {
            throw new IllegalArgumentException("Target Permission class cannot be a
DeniedPermission itself.");
        }
        Constructor<?> targetCtor;
        try {
            if (targetActions == null) {
                targetCtor = targetClass.getConstructor(String.class);
            }
            else {
                targetCtor = targetClass.getConstructor(String.class, String.class);
            }
        }
        catch (NoSuchMethodException nsme) {
            throw new IllegalArgumentException(MessageFormat.format(
                "Target Permission class [{0}] does not provide or expose a (String name)
or (String name, String actions) constructor.",
                targetClassName));
        }
        try {
            return (Permission) targetCtor
                .newInstance(((targetCtor.getParameterCount() == 1) ? new Object[] {
targetName }
                    : new Object[] { targetName, targetActions }));
        }
        catch (ReflectiveOperationException roe) {
            if (roe instanceof InvocationTargetException) {
                if (targetName == null) {
                    targetName = "<null>";
                }
                else if (targetName.trim().isEmpty()) {
                    targetName = "<empty>";
                }
                if (targetActions == null) {
                    targetActions = "<null>";
                }
            }
        }
    }

```



```

        else if (targetActions.trim().isEmpty()) {
            targetActions = "<empty>";
        }
        throw new IllegalArgumentException(MessageFormat.format(
            "Could not instantiate target Permission class [{0}]; provided target
name [{1}] and/or target actions [{2}] potentially erroneous.",
            targetClassName, targetName, targetActions), roe);
    }
    throw new RuntimeException(
        "Could not instantiate target Permission class [{0}]; an unforeseen error
occurred - see attached cause for details",
        roe);
    }
}

/**
 * Checks whether the given permission is implied by this one, as per the {@link
DeniedPermission
 * overview}.
 */
@Override
public boolean implies(Permission p) {
    if (p instanceof DeniedPermission) {
        return target.implies(((DeniedPermission) p).target);
    }
    return target.implies(p);
}

/**
 * Returns this denied permission's target permission (the actual positive permission
which is not
 * to be granted).
 */
public Permission getTargetPermission() {
    return target;
}
}

```

DenyingPolicy クラス

```

package com.example;

import java.security.CodeSource;
import java.security.NoSuchAlgorithmException;
import java.security.Permission;
import java.security.PermissionCollection;
import java.security.Policy;
import java.security.ProtectionDomain;
import java.security.UnresolvedPermission;
import java.util.Enumeration;

/**
 * Wrapper that adds rudimentary {@link DeniedPermission} processing capabilities to the
standard
 * file-backed <code>Policy</code>.
 */
public final class DenyingPolicy extends Policy {

```

```

{
    try {
        defaultPolicy = Policy.getInstance("javaPolicy", null);
    }
    catch (NoSuchAlgorithmException nsae) {
        throw new RuntimeException("Could not acquire default Policy.", nsae);
    }
}

private final Policy defaultPolicy;

@Override
public PermissionCollection getPermissions(CodeSource codesource) {
    return defaultPolicy.getPermissions(codesource);
}

@Override
public PermissionCollection getPermissions(ProtectionDomain domain) {
    return defaultPolicy.getPermissions(domain);
}

/**
 * @return
 *      <ul>
 *      <li><code>true</code> if:</li>
 *      <ul>
 *      <li><code>permission</code> <em>is not</em> an instance of
 *      <code>DeniedPermission</code>,</li>
 *      <li>an <code>implies(domain, permission)</code> invocation on the system-
default
 *      <code>Policy</code> yields <code>true</code>, and</li>
 *      <li><code>permission</code> <em>is not</em> implied by any
<code>DeniedPermission</code>s
 *      having potentially been assigned to <code>domain</code>.</li>
 *      </ul>
 *      <li><code>false</code>, otherwise.
 *      </ul>
 */
@Override
public boolean implies(ProtectionDomain domain, Permission permission) {
    if (permission instanceof DeniedPermission) {
        /*
         * At the policy decision level, DeniedPermissions can only themselves imply, not
be implied (as
         * they take away, rather than grant, privileges). Furthermore, clients aren't
supposed to use this
         * method for checking whether some domain _does not_ have a permission (which is
what
         * DeniedPermissions express after all).
         */
        return false;
    }

    if (!defaultPolicy.implies(domain, permission)) {
        // permission not granted, so no need to check whether denied
        return false;
    }

    /*
     * Permission granted--now check whether there's an overriding DeniedPermission. The
following

```

```

        * assumes that previousPolicy is a sun.security.provider.PolicyFile (different
implementations
        * might not support #getPermissions(ProtectionDomain) and/or handle
UnresolvedPermissions
        * differently).
        */

Enumeration<Permission> perms = defaultPolicy.getPermissions(domain).elements();
while (perms.hasMoreElements()) {
    Permission p = perms.nextElement();
    /*
        * DeniedPermissions will generally remain unresolved, as no code is expected to
check whether other
        * code has been "granted" such a permission.
        */
    if (p instanceof UnresolvedPermission) {
        UnresolvedPermission up = (UnresolvedPermission) p;
        if (up.getUnresolvedType().equals(DeniedPermission.class.getName())) {
            // force resolution
            defaultPolicy.implies(domain, up);
            // evaluate right away, to avoid reiterating over the collection
            p = new DeniedPermission(up.getUnresolvedName());
        }
    }
    if (p instanceof DeniedPermission && p.implies(permission)) {
        // permission denied
        return false;
    }
    // permission granted
    return true;
}

@Override
public void refresh() {
    defaultPolicy.refresh();
}

}

```

デモ

```

package com.example;

import java.security.Policy;

public class Main {

    public static void main(String... args) {
        Policy.setPolicy(new DenyingPolicy());
        System.setSecurityManager(new SecurityManager());
        // should fail
        System.getProperty("foo.bar");
    }

}

```

いくつかのをりてる

```
grant codeBase "file:///path/to/classes/bin/-"  
    permission java.util.PropertyPermission "*", "read,write";  
    permission com.example.DeniedPermission "java.util.PropertyPermission:foo.bar:read";  
};
```

に、Mainをして、ルール DeniedPermission が grant PropertyPermission をオーバーライドするため、することをします。されたパーミッションは "read"アクションだけをカバーし、 "foo.bar"プロパティだけをうため、setProperty("foo.baz", "xyz")びしはしたことにしてください。

オンラインでをむ <https://riptutorial.com/ja/java/topic/5712/>

143: れクラスとクラス

き

Javaをすると、はのクラスでクラスをすることができます。そのようなクラスはネストされたクラスとばれます。ネストされたクラスは、であるとされたはクラスとばれ、そうでないはなネストされたクラスとばれます。このページは、Javaのれクラスとクラスのをし、をするためのものです。

- パブリッククラスOuterClass {public class InnerClass {}} //インナークラスはprivateでもかまいません
- パブリッククラスOuterClass {public static class StaticNestedClass {}} //にネストされたクラスもプライベートにすることができます
- public void method{private class LocalClass {}} //ローカルクラスはにprivateです。
- SomeClass anonymousClassInstance = new SomeClass{}; //のクラスにはをけることができないため、アクセスはわしいです。 'SomeClass'がクラスの、ボディはすべてのメソッドをするがあります。
- SomeInterface anonymousClassInstance = しいSomeInterface{}; //ボディはすべてのインターフェイスメソッドをするがあります。

と

JavaJLSは、さまざまなJavaクラスをののようにします。

トップレベルのクラスは、ネストされたクラスではないクラスです。

ネストされたクラスは、のクラスまたはインタフェースののがするのクラスです。

クラスは、にまたはにされていないネストされたクラスです。

クラスは、メンバークラス、ローカルクラス、またはクラスです。インターフェイスのメンバクラスはにであるため、クラスとはみなされません。

にプログラマーは、クラスを「クラス」としてむレベルのクラスをします。また、ネストされたクラスのみをまたはにするために「ネストされたクラス」をするがあります。

のクラスとlambdaはなにありますが、lambdasはクラスです。

のい

- トップレベルのクラスは「ケース」です。これらは、アクセスのセマンティクスについたのとなるプログラムののにもえます。クラスの、するコンストラクタがアクセスについてされるコードでインスタンスできます。

- なネストされたクラスは、の2つのをいて、トップレベルのクラスとじアクセスおよびインスタンスのルールにいます。
 - ネストされたクラスは`private`クラスとしてすることができ、そのクラスをむトップレベルクラスのではアクセスできなくなります。
 - ネストされたクラスは、んでいるトップレベルクラスの`private`メンバーとテストみのクラスにアクセスできます。

なネストされたクラスは、なにの「エンティティタイプ」をするがあるにです。たとえば、ネストしたクラスをして「の」をす

- クラスは、みスコープでされたにアクセスするをします。
 - メンバークラスはインスタンスをできます。
 - メソッドでされたローカルクラスは、メソッドのローカルをすることもできますただし、それらが`final`。Java 8では、ににできます。
 - のクラスは、クラスまたはメソッドですることができ、じにってにアクセスできます
 -

クラス・インスタンスが、むクラス・インスタンスのをできるというのは、インスタンスにをちます。には、クラスのインスタンスがされるときに、するインスタンスをまたはにするがあります。

Examples

ネストされたクラスをしたなスタック

```
public class IntStack {

    private IntStackNode head;

    // IntStackNode is the inner class of the class IntStack
    // Each instance of this inner class functions as one link in the
    // Overall stack that it helps to represent
    private static class IntStackNode {

        private int val;
        private IntStackNode next;

        private IntStackNode(int v, IntStackNode n) {
            val = v;
            next = n;
        }
    }

    public IntStack push(int v) {
        head = new IntStackNode(v, head);
        return this;
    }

    public int pop() {
```

```

        int x = head.val;
        head = head.next;
        return x;
    }
}

```

そして、にネストされたクラスのをくめない、その。

```

public class Main {
    public static void main(String[] args) {

        IntStack s = new IntStack();
        s.push(4).push(3).push(2).push(1).push(0);

        //prints: 0, 1, 2, 3, 4,
        for(int i = 0; i < 5; i++) {
            System.out.print(s.pop() + ", ");
        }
    }
}

```

クラスとクラス

ネストされたクラスをするとき、そのネストされたクラスをにするにします

```

public class OuterClass1 {

    private static class StaticNestedClass {

    }

}

```

または

```

public class OuterClass2 {

    private class NestedClass {

    }

}

```

そのコアでは、ネストされたクラスは、クラスののインスタンスをたないのにし、なネストされたクラスはそうにします。これは、ネストされたクラスをインスタンスすることがされているとそのと、そのネストされたクラスのインスタンスにアクセスがされているのにします。のにえて

```

public class OuterClass1 {

    private int aField;
    public void aMethod(){}
}

```

```

private static class StaticNestedClass {
    private int innerField;

    private StaticNestedClass() {
        innerField = aField; //Illegal, can't access aField from static context
        aMethod();           //Illegal, can't call aMethod from static context
    }

    private StaticNestedClass(OuterClass1 instance) {
        innerField = instance.aField; //Legal
    }

}

public static void aStaticMethod() {
    StaticNestedClass s = new StaticNestedClass(); //Legal, able to construct in static
context
    //Do stuff involving s...
}

}

public class OuterClass2 {

    private int aField;

    public void aMethod() {}

    private class NestedClass {
        private int innerField;

        private NestedClass() {
            innerField = aField; //Legal
            aMethod(); //Legal
        }
    }

    public void aNonStaticMethod() {
        NestedClass s = new NestedClass(); //Legal
    }

    public static void aStaticMethod() {
        NestedClass s = new NestedClass(); //Illegal. Can't construct without surrounding
OuterClass2 instance.
//As this is a static context, there is no
surrounding OuterClass2 instance
    }
}

```

したがって、とのなは、にクラスのフィールドとメソッドにアクセスできるがあるかどうかによってなりますが、ネストされたクラスをいつどこでできるかにもします。

として、クラスのフィールドやメソッドにアクセスするがないかぎり、ネストされたクラスをにしてください。がなをいて、フィールドをにするののに、インスタンスへのアクセスをしないことでネストされたクラスでなをらし、エラーのをらしめます。

クラスのアクセス

Javaにおけるアクセスのなは、ここでつけることができます。しかし、らはどのようにInnerクラスとやりとりしますか

`public`は、いつものように、にアクセスできるスコープにのアクセスをえます。

```
public class OuterClass {

    public class InnerClass {

        public int x = 5;

    }

    public InnerClass createInner() {
        return new InnerClass();
    }
}

public class SomeOtherClass {

    public static void main(String[] args) {
        int x = new OuterClass().createInner().x; //Direct field access is legal
    }
}
```

`protected`と`default`もしていないのが、ネストされていないクラスにしてもじようにします。

いことに、`private`、それがするクラスにされません。むしろ、コンパイル`.java`ファイルにされます。つまり、OuterクラスはInnerクラスのフィールドとメソッドにフルアクセスできますが、`private`クラスとマークされていてもできます。

```
public class OuterClass {

    public class InnerClass {

        private int x;
        private void anInnerMethod() {}
    }

    public InnerClass aMethod() {
        InnerClass a = new InnerClass();
        a.x = 5; //Legal
        a.anInnerMethod(); //Legal
        return a;
    }
}
```

インナークラスは、`public`のをつことができます。それを`private`またはのきアクセスにマークすることにより、ののクラスはそのをインポートしてりてることができません。しかし、それらはとしてそのタイプのオブジェクトへのをることができます。

```
public class OuterClass {

    private class InnerClass{}
}
```

```

    public InnerClass makeInnerClass() {
        return new InnerClass();
    }
}

public class AnotherClass {

    public static void main(String[] args) {
        OuterClass o = new OuterClass();

        InnerClass x = o.makeInnerClass(); //Illegal, can't find type
        OuterClass.InnerClass x = o.makeInnerClass(); //Illegal, InnerClass has visibility
private
        Object x = o.makeInnerClass(); //Legal
    }
}

```

のクラス

のクラスは、され、のステートメントでインスタンスされるクラスのです。として、プログラムのどこかでできるクラスのはありません。つまり、それはです。

クラスは、クラスをしてパラメータとしてせるようにするがあるにします。これは、、インタフェースでわれます。えは

```

public static Comparator<String> CASE_INSENSITIVE =
    new Comparator<String>() {
        @Override
        public int compare(String string1, String string2) {
            return string1.toUpperCase().compareTo(string2.toUpperCase());
        }
    };

```

このクラスは、とのいをして2つのをする `Comparator<String>` オブジェクト `CASE_INSENSITIVE` をします。

クラスをしてにされインスタンスされるそののインタフェースは、 `Runnable` および `Callable` です。えは

```

// An anonymous Runnable class is used to provide an instance that the Thread
// will run when started.
Thread t = new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello world");
    }
});
t.start(); // Prints "Hello world"

```

のクラスは、クラスにづくこともできます。この、クラスはにのクラスを `extends` します。されるクラスがクラスの、クラスはすべてのメソッドをするがあります。メソッドをオーバーライドすることもできます。

コンストラクタ

クラスはなコンストラクタをつことはできません。わりに、されているクラスのコンストラクタにパラメータをすために`super(...)`をするのコンストラクタがされています。えは

```
SomeClass anon = new SomeClass(1, "happiness") {
    @Override
    public int someMethod(int arg) {
        // do something
    }
};
```

`SomeClass` サブクラスののコンストラクタは、びしグネチャ `SomeClass(int, String)` とする `SomeClass` コンストラクタをびします。コンストラクタができないは、コンパイルエラーがします。したコンストラクタによってスローされたも、のコンストラクタによってスローされます。

もちろん、これはインターフェイスをするときにはしません。インターフェイスからクラスをする、クラススーパークラスはなしのコンストラクタのみをつ `java.lang.Object` です。

メソッドローカルのクラス

メソッドローカルインナークラスとばれるメソッドにされたクラス。その、クラスのスコープはメソッドでされます。

メソッドローカルのクラスは、クラスがされているメソッドでのみインスタンスできます。

メソッドローカルクラスの

```
public class OuterClass {
    private void outerMethod() {
        final int outerInt = 1;
        // Method Local Inner Class
        class MethodLocalInnerClass {
            private void print() {
                System.out.println("Method local inner class " + outerInt);
            }
        }
        // Accessing the inner class
        MethodLocalInnerClass inner = new MethodLocalInnerClass();
        inner.print();
    }

    public static void main(String args[]) {
        OuterClass outer = new OuterClass();
        outer.outerMethod();
    }
}
```

すると、がられます。 Method local inner class 1。

なクラスからクラスにアクセスする

クラスへののはクラスをし`this`

```
public class OuterClass {
    public class InnerClass {
        public void method() {
            System.out.println("I can access my enclosing class: " + OuterClass.this);
        }
    }
}
```

クラスのフィールドとメソッドにアクセスできます。

```
public class OuterClass {
    private int counter;

    public class InnerClass {
        public void method() {
            System.out.println("I can access " + counter);
        }
    }
}
```

しかし、ののは、クラスをできます。

```
public class OuterClass {
    private int counter;

    public class InnerClass {
        private int counter;

        public void method() {
            System.out.println("My counter: " + counter);
            System.out.println("Outer counter: " + OuterClass.this.counter);

            // updating my counter
            counter = OuterClass.this.counter;
        }
    }
}
```

からなクラスのインスタンスをする

このクラスからも、のクラスからえるクラスをすることができます。

クラスはクラスにし、そのインスタンスへのをとします。クラスのインスタンスをするには、`new`はクラスのインスタンスにしてのみびすがあります。

```
class OuterClass {

    class InnerClass {
    }

}

class OutsideClass {
```

```
OuterClass outer = new OuterClass();

OuterClass.InnerClass createInner() {
    return outer.new InnerClass();
}
}
```

はouter.newしてouter.new。

オンラインでれクラスとクラスをむ <https://riptutorial.com/ja/java/topic/3317/れクラスとクラス>

144:

き

は、メソッドがをびすときにします。このようなはとばれている。なは、のなよりもであるがあります。しかし、いの、はスレッドののスタックをすることはありません。

このトピックには、Javaでののがまれています。

メソッドの

メソッドをするときは、のことがであることにしてください。

- 。これにより、がいつをしてするかをします。ののケースはのとおります。

```
if (n <= 1) {  
    return 1;  
}
```

- びし。このステートメントでは、されたパラメータでメソッドをびしします。ののびしはのとおります。

```
else {  
    return n * factorial(n - 1);  
}
```

このでは、nのをします。のはのとおります。

$$0 = 1$$

$$1 = 1$$

$$2 = 1 \times 2 = 2$$

$$3 = 1 \times 2 \times 3 = 6$$

$$4 = 1 \times 2 \times 3 \times 4 = 24$$

...

Javaとテールコールの

のJavaコンパイラJava 9までは、テールコールをしません。これはアルゴリズムのパフォーマンス

スにをえるがあり、がにいい、`StackOverflowError`クラッシュするがあります。 [いがJavaでになる](#)

Examples

のなえ

とはですか

に、は、がまたはにそれをびすときです。えは

```
// This method calls itself "infinitely"
public void useless() {
    useless(); // method calls itself (directly)
}
```

へのをするための

をしてのをするには、の2つのがあります。

1. のがなければなりません。これはのになります。がにすると、そののいびしはわれません。
2. レベルのは、よりさなをみるべきです。したがって、はをよりさいにします。がであるとすると、がすることがされます。

Javaでは、3のがあります。をするにはあまりにもくするはありません。 [いがJavaでになる](#)

のは、をしてをします。メソッド `factorial` がでどのようにをびすかにしてください。をびすたびに、パラメータ `n` を1らしめます `n` が1になるとベース、はそれくしません。

```
public int factorial(int n) {
    if (n <= 1) { // the base condition
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

これはオーバーフロー、または `n` きなにするコールスタックオーバーフローすなわち `StackOverflowError` をしていないため、Javaのをするなではありません。

Nのフィボナッチの

のメソッドは、をしてNのフィボナッチをします。

```
public int fib(final int n) {
    if (n > 2) {
        return fib(n - 2) + fib(n - 1);
    }
    return 1;
}
```

メソッドは、ケース $n \leq 2$ とケース $n > 2$ をします。これは、をするためにをすることをしてい

しかしながら、このはなものであるが、である。すなわち、メソッドのインスタンスは、を2びす
ため、 N がするにつれてがびされるがにする。のは $O(2^N)$ ですが、のはさ ON をちます。さらに、 O
 N でできる「じた」のがあります。

1からNまでののを

のメソッドは、をして0からNまでののをします。

```
public int sum(final int n) {
    if (n > 0) {
        return n + sum(n - 1);
    } else {
        return n;
    }
}
```

このは ON であり、テールコールをしてなループにらすことができます。、 $O(1)$ でをするじたのが
あります。

のNをする

のメソッドは、をして num のを exp のにしてします。

```
public long power(final int num, final int exp) {
    if (exp == 0) {
        return 1;
    }
    if (exp == 1) {
        return num;
    }
    return num * power(num, exp - 1);
}
```

メソッドは、をさせるケース2つのケース、 $n = 0$ および $n = 1$ と、そのメソッドをびすケースをし
ます。このは ON であり、テールコールをしてなループにらすことができます。

をしてをする

は、をにするためのなコードです

```
/**
 * Just a snippet to explain the idea of recursion
 *
 */

public class Reverse {
    public static void main (String args[]) {
        String string = "hello world";
```



```

        System.out.println(reverse(string)); //prints dlrow olleh
    }

    public static String reverse(String s) {
        if (s.length() == 1) {
            return s;
        }

        return reverse(s.substring(1)) + s.charAt(0);
    }
}

```

によるツリーデータのトラバース

のように、3つのメンバーデータ、のポインター、およびのポインターをつNodeクラスをえてみましょう。

```

public class Node {
    public int data;
    public Node left;
    public Node right;

    public Node(int data){
        this.data = data;
    }
}

```

のようにのNodeクラスのオブジェクトをしてしたツリーをすることができます。このトラバースは、ツリーのりのトラバースとされます。

```

public static void inOrderTraversal(Node root) {
    if (root != null) {
        inOrderTraversal(root.left); // traverse left sub tree
        System.out.print(root.data + " "); // traverse current node
        inOrderTraversal(root.right); // traverse right sub tree
    }
}

```

でしたように、をすると、りしアプローチではなデータをすることなく、 ツリーデータをトラバースすることができます。

のタイプ

は、メソッドびしのにじて、 **Head Recursion**または**Tail Recursion**のいずれかにできます。

では、びしは、がしたときにのののにますのやでこつたとえてください。

では、のことです。びしのにがわれます。 2つのスタイルのからすることはであるかもしれませんが、そのはすべてのいをむことができます。

パスののにのびしをつパスをつは、とばれるものをします。ののはをします。がであるとしたのことは、されたパラメータでをびすことです。パスのわりにのびしをつは、をしています。

```

public void tail(int n)
{
    if(n == 1)
        return;
    else
        System.out.println(n);

    tail(n-1);
}

public void head(int n)
{
    if(n == 0)
        return;
    else
        head(n-1);

    System.out.println(n);
}

```

びしがメソッドのにした、それはtail recursionとばれtail recursion。テールはsimilar to a loop。このmethod executes all the statements before jumping into the next recursive call。

びしがbeginning of a method, it is called a head recursionのbeginning of a method, it is called a head recursionする、びしbeginning of a method, it is called a head recursion。このmethod saves the state before jumping into the next recursive call。

headtailのい

StackOverflowErrorおよびループへの

びしが"すぎ"になると、StackOverflowErrorます。Javaはスレッドのスタックのすべてのメソッドびしにしいフレームをりてます。しかし、スレッドのスタックのスペースはられています。スタックのフレームがすぎると、スタックオーバーフローSOがします。

```

public static void recursion(int depth) {
    if (depth > 0) {
        recursion(depth-1);
    }
}

```

このメソッドをきなパラメータでびすとおそらくrecursion(50000)、スタックのオーバーフローがします。なはスレッドのスタックサイズ、スレッド、-Xssなどのコマンドラインパラメータ、またはJVMのデフォルトサイズ。

びしごとのデータをデータにすることで、をループにすることができます。このデータは、スレッドスタックではなくヒープにすることができます。

に、メソッドびしのをするためになデータはスタックにでき、whileループはびしを「シミュレート」するためにできます。なデータにはのがあります。

- メソッドがびされたオブジェクトインスタンスメソッドのみ
- メソッドのパラメータ
-
- またはメソッドのの

のクラスでは、のさまでツリーをにすることができます。

```

public class Node {

    public int data;
    public Node left;
    public Node right;

    public Node(int data) {
        this(data, null, null);
    }

    public Node(int data, Node left, Node right) {
        this.data = data;
        this.left = left;
        this.right = right;
    }

    public void print(final int maxDepth) {
        if (maxDepth <= 0) {
            System.out.print("(...)");
        } else {
            System.out.print("(");
            if (left != null) {
                left.print(maxDepth-1);
            }
            System.out.print(data);
            if (right != null) {
                right.print(maxDepth-1);
            }
            System.out.print(")");
        }
    }
}

```

えは

```

Node n = new Node(10, new Node(20, new Node(50), new Node(1)), new Node(30, new Node(42),
null));
n.print(2);
System.out.println();

```

プリント

```
((... )20(... )10(... )30))
```

これはのループにすることができます

```

public class Frame {

    public final Node node;

    // 0: before printing anything
    // 1: before printing data
    // 2: before printing ")"
    public int state = 0;
    public final int maxDepth;
}

```

```

public Frame(Node node, int maxDepth) {
    this.node = node;
    this.maxDepth = maxDepth;
}
}

```

```

List<Frame> stack = new ArrayList<>();
stack.add(new Frame(n, 2)); // first frame = initial call

while (!stack.isEmpty()) {
    // get topmost stack element
    int index = stack.size() - 1;
    Frame frame = stack.get(index); // get topmost frame
    if (frame.maxDepth <= 0) {
        // terminal case (too deep)
        System.out.print("(...)");
        stack.remove(index); // drop frame
    } else {
        switch (frame.state) {
            case 0:
                frame.state++;

                // do everything done before the first recursive call
                System.out.print("(");
                if (frame.node.left != null) {
                    // add new frame (recursive call to left and stop)
                    stack.add(new Frame(frame.node.left, frame.maxDepth - 1));
                    break;
                }
            case 1:
                frame.state++;

                // do everything done before the second recursive call
                System.out.print(frame.node.data);
                if (frame.node.right != null) {
                    // add new frame (recursive call to right and stop)
                    stack.add(new Frame(frame.node.right, frame.maxDepth - 1));
                    break;
                }
            case 2:
                // do everything after the second recursive call & drop frame
                System.out.print(")");
                stack.remove(index);
        }
    }
}
System.out.println();

```

これはなアプローチのなるです。しばしば、フレームをする、および/またはフレームデータをす、よりいをえすことができます。

Javaでいがになる

をして2つののをするためののをえてみましょう。

```

public static int add(int a, int b) {

```

```

    if (a == 0) {
        return b;
    } else {
        return add(a - 1, b + 1); // TAIL CALL
    }
}

```

これはアルゴリズムにはいいですが、きながあります。きな `a add` って `add` をびすと、`StackOverflowError`、なくともJava 9のJavaバージョンでクラッシュします。

なプログラミングおよびのくのでは、コンパイラはをします。コンパイラは`add` タグきで `add` のびしがテールコールであり、ループとしてをにきえることにくでしょう。このはテールコールとばれます。

しかし、のJavaコンパイラは、テール・コールをしません。わりに、`add` びしによってスレッドのスタックにしいフレームがりてられます。たとえば、`add(1000, 1)` びすと、`1000` びしがあり、`1001` のになります。

は、スレッドがされるときにJavaスレッドスタックのサイズがされることです。これには、シングルスレッドプログラムの「メイン」スレッドもまれます。スタックフレームがすぎると、スタックがオーバーフローします。JVMはこれをして`StackOverflowError` をスローします。

これにする1つのアプローチは、によりきなスタックをすることです。スタックのデフォルトサイズをするJVMオプションがあり、スタックサイズを`Thread` コンストラクタパラメータとしてすることもできます。ながら、これはスタックのオーバーフローを「め」ます。さらにきなスタックをとするをうがあるは、`StackOverflowError` がされます。

のは、いがするがあるアルゴリズムをし、ソースコードレベルでテールコールをですることです。たとえば、のように`add` メソッドをきえることができます。

```

public static int add(int a, int b) {
    while (a != 0) {
        a = a - 1;
        b = b + 1;
    }
    return b;
}

```

らかに、2つのをするよりいがあります。は、によるテールコールのをするためのものです。

なぜテールコールがJavaではまだされていないのですか

Javaへのテール・コールのをするのではありません。えば

- いくつかのコードは、`StackOverflowError` にしてえばののきさにを`StackOverflowError` ことができます。
- サンドボックスのセキュリティは、のないコードでアクションをするかどうかをするに、コールスタックをすることがしばしばにちます。

John Roseが「[VMのテールコール](#)」でしたように

びしのスタックフレームをすることによるは、いくつかのAPI、にアクセスコントロールのチェックやスタックトレースからることができ、びしのびしがびしをびしたかのように。ただし、びしメソッドのリンケージとアクセシビリティは、コントロールのにされ、びしのびしをにれます。

いえれば、テール・コールをすると、アクセスメソッドは、されたAPIができるコードによってびされているとってするがあります。

オンラインでをむ <https://riptutorial.com/ja/java/topic/914/>

145:

き

Java enums `enum` キーワードをしてされていますは、クラスのののです。

- `[public / protected / private] enum Enum_name { // しいをします。`
- `ENUM_CONSTANT_1 [、 ENUM_CONSTANT_2 ...]; // enumをします。これはのののでなければならず、にセミコロンをけてカンマでるがあります。`
- `ENUM_CONSTANT_1param[、 ENUM_CONSTANT_2param...]; // パラメータでenumをします。パラメータはコンストラクタとするがあります。`
- `ENUM_CONSTANT_1 {...} [、 ENUM_CONSTANT_2 {...} ...]; // オーバーライドされたメソッドでをします。enumにメソッドがまれているは、これをうがあります。そのようなはすべてされなければならない。`
- `ENUM_CONSTANT.name // enumののStringをします。`
- `ENUM_CONSTANT.ordinal // こののをします。そののは、にゼロのがされます。`
- `Enum_name.values // びされるたびにそののすべてのをむしいEnum_name []をします。`
- `Enum_name.valueOf "ENUM_CONSTANT" // ENUM_CONSTANT.nameの - えられたのをします。`
- `Enum.valueOfEnum_name.class、 "ENUM_CONSTANT" // のものとENUM_CONSTANT.nameの - えられたのをします。`

はに `java.lang.Enum` をしているため、クラスをするはです。ただし、くのインターフェイスをできません。

ヒントとコツ

らのなのために、enumをキーとしてできるよりな **マップ** と **セット** があります。これらは、のカウンターパートよりもにくされます。

Examples

との

`Enum` は、のをするためにコンパイルにられているかだけインスタンスされるクラスのシュガーとえることができます。

さまざまなをするなは、のようにされます。

```
public enum Season {  
    WINTER,  
    SPRING,
```

```
SUMMER,  
FALL  
}
```

はずしものであるはありませんが、のはにで、はアンダースコアでられています。

Enumをのファイルにできます

```
/**  
 * This enum is declared in the Season.java file.  
 */  
public enum Season {  
    WINTER,  
    SPRING,  
    SUMMER,  
    FALL  
}
```

しかし、のクラスのことすることもできます

```
public class Day {  
  
    private Season season;  
  
    public String getSeason() {  
        return season.name();  
    }  
  
    public void setSeason(String season) {  
        this.season = Season.valueOf(season);  
    }  
  
    /**  
     * This enum is declared inside the Day.java file and  
     * cannot be accessed outside because it's declared as private.  
     */  
    private enum Season {  
        WINTER,  
        SPRING,  
        SUMMER,  
        FALL  
    }  
  
}
```

に、メソッドまたはコンストラクタでEnumをすることはできません。

```
public class Day {  
  
    /**  
     * Constructor  
     */  
    public Day() {  
        // Illegal. Compilation error  
        enum Season {  

```



```

        WINTER,
        SPRING,
        SUMMER,
        FALL
    }
}

public void aSimpleMethod() {
    // Legal. You can declare a primitive (or an Object) inside a method. Compile!
    int primitiveInt = 42;

    // Illegal. Compilation error.
    enum Season {
        WINTER,
        SPRING,
        SUMMER,
        FALL
    }

    Season season = Season.SPRING;
}
}

```

することはできません

```

public enum Season {
    WINTER,
    WINTER, //Compile Time Error : Duplicate Constants
    SPRING,
    SUMMER,
    FALL
}

```

enumのすべてののは、デフォルトで**public**、**static**および**final**です。すべてののは**static**なので、のをしてアクセスできます。

は、メソッドパラメータとしてすことができます

```

public static void display(Season s) {
    System.out.println(s.name()); // name() is a built-in method that gets the exact name of
    the enum constant
}

display(Season.WINTER); // Prints out "WINTER"

```

`values()` メソッドをして、のができます。は、されるのになることがされています。

```

Season[] seasons = Season.values();

```

このメソッドは、びされるたびにしいのをりてます。

をするには

```
public static void enumIterate() {
    for (Season s : Season.values()) {
        System.out.println(s.name());
    }
}
```

switchでenumをできます。

```
public static void enumSwitchExample(Season s) {
    switch(s) {
        case WINTER:
            System.out.println("It's pretty cold");
            break;
        case SPRING:
            System.out.println("It's warming up");
            break;
        case SUMMER:
            System.out.println("It's pretty hot");
            break;
        case FALL:
            System.out.println("It's cooling down");
            break;
    }
}
```

==をしてenumをすることもできます。

```
Season.FALL == Season.WINTER    // false
Season.SPRING == Season.SPRING  // true
```

をするもう1つのは、のようにとしににすることができるので、いとみなされるのよう`equals()`をうことです

```
Season.FALL.equals(Season.FALL); // true
Season.FALL.equals(Season.WINTER); // false
Season.FALL.equals("FALL"); // false and no compiler error
```

のインスタンスのがまた、enumにすることができないのクラスとに、ので、インスタンスがにでないenumにされているようにフィールドをむことができます。

```
public enum MutableExample {
    A,
    B;

    private int count = 0;

    public void increment() {
        count++;
    }

    public void print() {
```

```

        System.out.println("The count of " + name() + " is " + count);
    }
}

// Usage:
MutableExample.A.print();           // Outputs 0
MutableExample.A.increment();
MutableExample.A.print();           // Outputs 1 -- we've changed a field
MutableExample.B.print();           // Outputs 0 -- another instance remains unchanged

```

しかし、いは、`enum` インスタンスをにすることです。つまり、のフィールドをたないや、すべてのフィールドが`final`とマークされているのです。これにより、アプリケーションの、`enum`はメモリをリークさせず、すべてのスレッドにわたってそのインスタンスをすることがであることをします。

`Enum` クラスはのことをうため、`Enum` はに `Serializable` と `Comparable` します。

```

public abstract class Enum<E extends Enum<E>>
    extends Object
    implements Comparable<E>, Serializable

```

コンストラクタをつ

`enum` は `public` コンストラクタをつことはできません。ただし、`private` コンストラクタはけられます `enum` のコンストラクタはデフォルトで `package-private` です。

```

public enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25); // usual names for US coins
    // note that the above parentheses and the constructor arguments match
    private int value;

    Coin(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}

int p = Coin.NICKEL.getValue(); // the int value will be 5

```

のインスタンスはられているので、すべてのフィールドをにしてゲッターメソッドをすることをおめします。

`Enum` を `class` としてする、のようになります。

```

public class Coin<T extends Coin<T>> implements Comparable<T>, Serializable{
    public static final Coin PENNY = new Coin(1);
    public static final Coin NICKEL = new Coin(5);
    public static final Coin DIME = new Coin(10);

```

```

public static final Coin QUARTER = new Coin(25);

private int value;

private Coin(int value){
    this.value = value;
}

public int getValue() {
    return value;
}
}

int p = Coin.NICKEL.getValue(); // the int value will be 5

```

Enumはにであるため、enumのをするためにsetterをすることができます。しかし、これはいいとえられ、けるべきです。

ベストプラクティスは、Enumフィールドを`final`にすることです。

```

public enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);

    private final int value;

    Coin(int value){
        this.value = value;
    }

    ...

}

```

じでのコンストラクタをすることができます。あなたがするとき、あなたのenumでは、びされるコンストラクタをします

```

public enum Coin {
    PENNY(1, true), NICKEL(5, false), DIME(10), QUARTER(25);

    private final int value;
    private final boolean isCopperColored;

    Coin(int value){
        this(value, false);
    }

    Coin(int value, boolean isCopperColored){
        this.value = value;
        this.isCopperColored = isCopperColored;
    }

    ...

}

```

EnumクラスはSerializableをするがあります。

メソッドとスタティックブロックの

には、すべてのクラスとに、メソッドをめることができます。これがどのようにするかをするために、のようなenumをします

```
public enum Direction {  
    NORTH, SOUTH, EAST, WEST;  
}
```

をにすメソッドをしましょう

```
public enum Direction {  
    NORTH, SOUTH, EAST, WEST;  
  
    public Direction getOpposite(){  
        switch (this){  
            case NORTH:  
                return SOUTH;  
            case SOUTH:  
                return NORTH;  
            case WEST:  
                return EAST;  
            case EAST:  
                return WEST;  
            default: //This will never happen  
                return null;  
        }  
    }  
}
```

これは、フィールドとイニシャライザブロックをすることでさらにできます。

```
public enum Direction {  
    NORTH, SOUTH, EAST, WEST;  
  
    private Direction opposite;  
  
    public Direction getOpposite(){  
        return opposite;  
    }  
  
    static {  
        NORTH.opposite = SOUTH;  
        SOUTH.opposite = NORTH;  
        WEST.opposite = EAST;  
        EAST.opposite = WEST;  
    }  
}
```

このでは、をプライベートインスタンスフィールドにされているoppositeめてされ、Directionされます。ので、こののケースではNORTH SOUTHとに、々がすることはできません **コンストラクタ** **つ** ここでコンストラクタ NORTH (SOUTH), SOUTH (NORTH), EAST (WEST), WEST (EAST) よりエレガントにな

とできるようになり `opposite` します `final` にされるが、となり、したがってされない。

インターフェースをする

これは、プリコンパイルされたパターンにして `String` をテストするびしなでもある `enum` です。

```
import java.util.function.Predicate;
import java.util.regex.Pattern;

enum RegEx implements Predicate<String> {
    UPPER("[A-Z]+"), LOWER("[a-z]+"), NUMERIC("[+-]?[0-9]+");

    private final Pattern pattern;

    private RegEx(final String pattern) {
        this.pattern = Pattern.compile(pattern);
    }

    @Override
    public boolean test(final String input) {
        return this.pattern.matcher(input).matches();
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println(RegEx.UPPER.test("ABC"));
        System.out.println(RegEx.LOWER.test("abc"));
        System.out.println(RegEx.NUMERIC.test("+111"));
    }
}
```

のメンバーは、このメソッドをすることもできます。

```
import java.util.function.Predicate;

enum Acceptor implements Predicate<String> {
    NULL {
        @Override
        public boolean test(String s) { return s == null; }
    },
    EMPTY {
        @Override
        public boolean test(String s) { return s.equals(""); }
    },
    NULL_OR_EMPTY {
        @Override
        public boolean test(String s) { return NULL.test(s) || EMPTY.test(s); }
    };
}

public class Main {
    public static void main(String[] args) {
        System.out.println(Acceptor.NULL.test(null)); // true
        System.out.println(Acceptor.EMPTY.test("")); // true
        System.out.println(Acceptor.NULL_OR_EMPTY.test(" ")); // false
    }
}
```

パターン

このは、の「な」セットけれるがあるenumを、プログラマは、ののようなをすることができるclassここanywereされるインタフェースすることにより、enum Sをしなければならないが。

```
public interface ExtensibleEnum {  
    String name();  
}
```

このようにして、インタフェースによってタグけされたenumパラメータとしてすることができ、プログラマはメソッドによってけられるのenumをすることができます。これは、えは、デフォルトのなenumがあるAPIにおいてであり、これらのAPIのユーザは、よりくのでenumを「」したい。

デフォルトのenumのセットは、のようになります。

```
public enum DefaultValues implements ExtensibleEnum {  
    VALUE_ONE, VALUE_TWO;  
}
```

のはのようになります。

```
public enum ExtendedValues implements ExtensibleEnum {  
    VALUE_THREE, VALUE_FOUR;  
}
```

enumのをすサンプル - printEnum()がのenumのをけれるにしてください

```
private void printEnum(ExtensibleEnum val) {  
    System.out.println(val.name());  
}  
  
printEnum(DefaultValues.VALUE_ONE);    // VALUE_ONE  
printEnum(DefaultValues.VALUE_TWO);    // VALUE_TWO  
printEnum(ExtendedValues.VALUE_THREE); // VALUE_THREE  
printEnum(ExtendedValues.VALUE_FOUR);  // VALUE_FOUR
```

このパターンは、のでに1つのでされているをすることをげるものではありません。これらののはのインスタンスになります。また、switch-on-enumをすることはできません。これは、のenumではなくインターフェースであるからです。

メソッドをつ

はメソッドをすることができ、それぞれのenumメンバーはするがあります。

```
enum Action {  
    DODGE {  
        public boolean execute(Player player) {  
            return player.isAttacking();  
        }  
    }  
}
```

```

    }
},
ATTACK {
    public boolean execute(Player player) {
        return player.hasWeapon();
    }
},
JUMP {
    public boolean execute(Player player) {
        return player.getCoordinates().equals(new Coordinates(0, 0));
    }
};

public abstract boolean execute(Player player);
}

```

これにより、enumメンバは、のメソッドのをオンにすることなく、されたにしてのをできます。

このパターンは、および/またはインターフェイスのをしてされるもののいです。

enumをする

ずしもenumができるほどではない。enumをするには、のjavadocをします。

```

/**
 * United States coins
 */
public enum Coins {

    /**
     * One-cent coin, commonly known as a penny,
     * is a unit of currency equaling one-hundredth
     * of a United States dollar
     */
    PENNY(1),

    /**
     * A nickel is a five-cent coin equaling
     * five-hundredth of a United States dollar
     */
    NICKEL(5),

    /**
     * The dime is a ten-cent coin refers to
     * one tenth of a United States dollar
     */
    DIME(10),

    /**
     * The quarter is a US coin worth 25 cents,
     * one-fourth of a United States dollar
     */
    QUARTER(25);

    private int value;

    Coins(int value){
        this.value = value;
    }
}

```



```

    }

    public int getValue(){
        return value;
    }
}

```

のをする

enumクラスには、`values()` というののメソッドがまわって `values()`。このメソッドは、そのenumのすべてのものを返します。このメソッドをしてをすることができます。ただし、このメソッドは呼び出されるたびに新しいものを返すことにすることができます。

```

public enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;

    /**
     * Print out all the values in this enum.
     */
    public static void printAllDays() {
        for(Day day : Day.values()) {
            System.out.println(day.name());
        }
    }
}

```

Setがなは、`EnumSet.allOf(Day.class)` もできます。

パラメータとしての

ジェネリックのクラスをjavaにくるときには、typeパラメータがenumであることをすることができます。すべてののはEnumクラスをするので、のをすることができます。

```

public class Holder<T extends Enum<T>> {
    public final T value;

    public Holder(T init) {
        this.value = init;
    }
}

```

このでは、`T` はでなければなりません。

でをする

enum `DayOfWeek` があるとし `DayOfWeek`

```

enum DayOfWeek {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;
}

```

enumはみみのな `valueOf()` メソッドでコンパイルされます。このメソッドはでするためにできます

```
String dayName = DayOfWeek.SUNDAY.name();
assert dayName.equals("SUNDAY");

DayOfWeek day = DayOfWeek.valueOf(dayName);
assert day == DayOfWeek.SUNDAY;
```

これは、をしてもです。

```
Class<DayOfWeek> enumType = DayOfWeek.class;
DayOfWeek day = Enum.valueOf(enumType, "SUNDAY");
assert day == DayOfWeek.SUNDAY;
```

されたenumにするのがない、これらの `valueOf()` メソッドはどちらも `IllegalArgumentException` をスローします。

Guavaライブラリは、なをするためにGuava `Optional` をすヘルパーメソッド `Enums.getIfPresent()` をします。

```
DayOfWeek defaultDay = DayOfWeek.SUNDAY;
DayOfWeek day = Enums.valueOf(DayOfWeek.class, "INVALID").or(defaultDay);
assert day == DayOfWeek.SUNDAY;
```

でシングルトンパターンをする

は、がめてされるときにインスタンスされます。したがって、ので `Singleton` ソフトウェアパターンをすることができます。

```
public enum Attendant {

    INSTANCE;

    private Attendant() {
        // perform some initialization routine
    }

    public void sayHello() {
        System.out.println("Hello!");
    }
}

public class Main {

    public static void main(String... args) {
        Attendant.INSTANCE.sayHello();// instantiated at this point
    }
}
```

Joshua Blochの "Effective Java"のによれば、がシングルトンをするのです。このアプローチには

のがあります。

- スレッドの
- インスタンスの
- すぐにえるシリアル

また、 [インタフェース](#)をするセクションにすように、このシングルトンは1つのインタフェースをすることもできます。

プロパティフィールドをつ

たちは、だけでなく、よりくので `enum` をいたい、2つのをできるようにしたいとえています。

のをえてみましょう。

```
public enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);

    private final int value;

    Coin(int value){
        this.value = value;
    }

    public boolean isGreaterThan(Coin other){
        return this.value > other.value;
    }
}
```

ここでは、そのをす `Coin` という `Enum` をしました。 `isGreaterThan` メソッドをすると、2つの `enum` `isGreaterThan` をできます。

```
Coin penny = Coin.PENNY;
Coin dime = Coin.DIME;

System.out.println(penny.isGreaterThan(dime)); // prints: false
System.out.println(dime.isGreaterThan(penny)); // prints: true
```

をにする

を `String` にするは、2つのがあります。

々がっていると

```
public enum Fruit {
    APPLE, ORANGE, STRAWBERRY, BANANA, LEMON, GRAPE_FRUIT;
}
```

では、 `Fruit.APPLE` ようなものをどうすれば `"APPLE"` できますか

name() をしてする name()

name() は enum メソッドであり、enum の String をします。String は、enum がどのようにされたかをします。

えば

```
System.out.println(Fruit.BANANA.name()); // "BANANA"
System.out.println(Fruit.GRAPE_FRUIT.name()); // "GRAPE_FRUIT"
```

toString() をしてする

toString() は、デフォルトでは、name() とじをするようにオーバーライドされています。

しかし、toString() はによってオーバーライドされ、ユーザーフレンドリな String しやすくなります

あなたのコードをチェックしたいなら、toString() をしないでください toString() name() はもっています。をログやなどにするときのみ toString() します

デフォルトでは

```
System.out.println(Fruit.BANANA.toString()); // "BANANA"
System.out.println(Fruit.GRAPE_FRUIT.toString()); // "GRAPE_FRUIT"
```

きされる

```
System.out.println(Fruit.BANANA.toString()); // "Banana"
System.out.println(Fruit.GRAPE_FRUIT.toString()); // "Grape Fruit"
```

Enum のボディ

enum ののなをすることがである enum のデフォルトのオーバーライド enum 、このは、のとしてられています。

John、Ben、Luke の3のピアノのが PianoClass というの enum PianoClass でのようにされているとします。

```
enum PianoClass {
    JOHN, BEN, LUKE;
    public String getSex() {
```

```

        return "Male";
    }
    public String getLevel() {
        return "Beginner";
    }
}

```

そして12の、リタとトムがセックスとレベルで、のものとしな

```

enum PianoClass2 {
    JOHN, BEN, LUKE, RITA, TOM;
    public String getSex() {
        return "Male"; // issue, Rita is a female
    }
    public String getLevel() {
        return "Beginner"; // issue, Tom is an intermediate student
    }
}

```

のように、にしいをにするのはしいとはりません。

```

PianoClass2 tom = PianoClass2.TOM;
PianoClass2 rita = PianoClass2.RITA;
System.out.println(tom.getLevel()); // prints Beginner -> wrong Tom's not a beginner
System.out.println(rita.getSex()); // prints Male -> wrong Rita's not a male

```

PianoClass2デフォルトのをのようにオーバーライドするRitaとTomのそれぞれのをすることがで
きます

```

enum PianoClass3 {
    JOHN, BEN, LUKE,
    RITA {
        @Override
        public String getSex() {
            return "Female";
        }
    },
    TOM {
        @Override
        public String getLevel() {
            return "Intermediate";
        }
    };
    public String getSex() {
        return "Male";
    }
    public String getLevel() {
        return "Beginner";
    }
}

```

トムのレベルとリタのセックスは、のようになります。

```

PianoClass3 tom = PianoClass3.TOM;
PianoClass3 rita = PianoClass3.RITA;

```

```
System.out.println(tom.getLevel()); // prints Intermediate
System.out.println(rita.getSex()); // prints Female
```

コンテンツのボディをするのは、コンストラクタをすることです。

```
enum Friend {
    MAT("Male"),
    JOHN("Male"),
    JANE("Female");

    private String gender;

    Friend(String gender) {
        this.gender = gender;
    }

    public String getGender() {
        return this.gender;
    }
}
```

および

```
Friend mat = Friend.MAT;
Friend john = Friend.JOHN;
Friend jane = Friend.JANE;
System.out.println(mat.getGender()); // Male
System.out.println(john.getGender()); // Male
System.out.println(jane.getGender()); // Female
```

ゼロインスタンスの

```
enum Util {
    /* No instances */;

    public static int clamp(int min, int max, int i) {
        return Math.min(Math.max(i, min), max);
    }

    // other utility methods...
}
```

シングルトン 1インスタンスクラスにenumをできるように、ユーティリティクラス0インスタンスクラスにできます。enumののリストをずさせてください、。

のコンストラクタとして、プロトののためのインスタンスをぐためのゼロインスタンスのとプライベートなコンストラクタのをしてください。

フィールドをつ

enumクラスにフィールドがなは、enumのにされることにしてください。つまり、のコードではNullPointerExceptionます。

```
enum Example {
    ONE(1), TWO(2);

    static Map<String, Integer> integers = new HashMap<>();

    private Example(int value) {
        integers.put(this.name(), value);
    }
}
```

これをするはのとおりです。

```
enum Example {
    ONE(1), TWO(2);

    static Map<String, Integer> integers;

    private Example(int value) {
        putValue(this.name(), value);
    }

    private static void putValue(String name, int value) {
        if (integers == null)
            integers = new HashMap<>();
        integers.put(name, value);
    }
}
```

フィールドをしないでください

```
enum Example {
    ONE(1), TWO(2);

    // after initialisation integers is null!!
    static Map<String, Integer> integers = null;

    private Example(int value) {
        putValue(this.name(), value);
    }

    private static void putValue(String name, int value) {
        if (integers == null)
            integers = new HashMap<>();
        integers.put(name, value);
    }
    // !!this may lead to null pointer exception!!
    public int getValue(){
        return (Example.integers.get(this.name()));
    }
}
```

- のをする
 - として、putValueがびされ、がされます
- ながされます。
 - =ヌル; //のにされるので、のはわれます

Enumのとインクルード

Enumはのみをみ、==とできます。したがって、チェックのみがで、.equalsメソッドをするはありません。さらに、.equalsがしくされていない、NullPointerExceptionするがありますが、これは== checkのには==ません。

```
enum Day {
    GOOD, AVERAGE, WORST;
}

public class Test {

    public static void main(String[] args) {
        Day day = null;

        if (day.equals(Day.GOOD)) { //NullPointerException!
            System.out.println("Good Day!");
        }

        if (day == Day.GOOD) { //Always use == to compare enum
            System.out.println("Good Day!");
        }

    }
}
```

[EnumSet](#) クラスをグループ、するには、なるメソッドをむ [EnumSet](#) クラスがあります。

- EnumSet#range 2つのエンドポイントでされたでenumのサブセットをする
- EnumSet#of をたないのenumのセット。 ofメソッドがオーバーロードされています。
- EnumSet#complementOf メソッドのパラメータでされるenumをするenumのセット

```
enum Page {
    A1, A2, A3, A4, A5, A6, A7, A8, A9, A10
}

public class Test {

    public static void main(String[] args) {
        EnumSet<Page> range = EnumSet.range(Page.A1, Page.A5);

        if (range.contains(Page.A4)) {
            System.out.println("Range contains A4");
        }

        EnumSet<Page> of = EnumSet.of(Page.A1, Page.A5, Page.A3);

        if (of.contains(Page.A1)) {
            System.out.println("Of contains A1");
        }

    }
}
```


オンラインでをむ <https://riptutorial.com/ja/java/topic/155/>

146: メソッドディスパッチ

き

ダイナミックメソッドディスパッチとはですか

メソッドディスパッチは、オーバーライドされたメソッドへのびしがコンパイルではなくにされるプロセスです。オーバーライドされたメソッドがによってびされると、Javaはするオブジェクトのについてするメソッドのバージョンをします。これは、ランタイムポリモーフィズムとしてもられています。

これををいてていきます。

- バインディング=バインディング
- クラスはインスタンスできませんが、サブクラスすることができますクラスのはBase
- メソッドは、なしでされるメソッドです
- クラスには、のにかかわらずされたメソッドがすることがあります
- クラスがサブクラスされている、サブクラスは、そのクラスのすべてのメソッドのをします。しかし、そうでなければ、そのサブクラスもabstractとしなければならない
- メソッドディスパッチは、オーバーライドされたメソッドへのびしをにするメカニズムです。これは、javaがランタイムポリモーフィズムをするです。
- アップキャストUppercastingサブタイプをスーパータイプにキャストし、ツリーまできにします。
- ランタイム=

Examples

メソッドディスパッチ - サンプルコード

クラス

```
package base;

/*
Abstract classes cannot be instantiated, but they can be subclassed
*/
public abstract class ClsVirusScanner {

    //With One Abstract method
    public abstract void fnStartScan();

    protected void fnCheckForUpdateVersion(){
        System.out.println("Perform Virus Scanner Version Check");
    }

    protected void fnBootTimeScan() {
```

```

        System.out.println("Perform BootTime Scan");
    }
    protected void fnInternetSecutiry(){
        System.out.println("Scan for Internet Security");
    }

    protected void fnRealTimeScan(){
        System.out.println("Perform RealTime Scan");
    }

    protected void fnVirusMalwareScan(){
        System.out.println("Detect Virus & Malware");
    }
}

```

クラスのメソッドのオーバーライド

```

import base.ClsVirusScanner;

//All the 3 child classes inherits the base class ClsVirusScanner
//Child Class 1
class ClsPaidVersion extends ClsVirusScanner{
    @Override
    public void fnStartScan() {
        super.fnCheckForUpdateVersion();
        super.fnBootTimeScan();
        super.fnInternetSecutiry();
        super.fnRealTimeScan();
        super.fnVirusMalwareScan();
    }
}; //ClsPaidVersion IS-A ClsVirusScanner
//Child Class 2

class ClsTrialVersion extends ClsVirusScanner{
    @Override
    public void fnStartScan() {
        super.fnInternetSecutiry();
        super.fnVirusMalwareScan();
    }
}; //ClsTrialVersion IS-A ClsVirusScanner

//Child Class 3
class ClsFreeVersion extends ClsVirusScanner{
    @Override
    public void fnStartScan() {
        super.fnVirusMalwareScan();
    }
}; //ClsTrialVersion IS-A ClsVirusScanner

```

ダイナミック/レイトバインディングは、メソッドディスパッチにつながります。

```

//Calling Class
public class ClsRunTheApplication {

    public static void main(String[] args) {

        final String VIRUS_SCANNER_VERSION = "TRIAL_VERSION";
    }
}

```

```

//Parent Refers Null
ClsVirusScanner objVS=null;

//String Cases Supported from Java SE 7
switch (VIRUS_SCANNER_VERSION){
case "FREE_VERSION":

    //Parent Refers Child Object 3
    //ClsFreeVersion IS-A ClsVirusScanner
    objVS = new ClsFreeVersion(); //Dynamic or Runtime Binding
    break;
case "PAID_VERSION":

    //Parent Refers Child Object 1
    //ClsPaidVersion IS-A ClsVirusScanner
    objVS = new ClsPaidVersion(); //Dynamic or Runtime Binding
    break;
case "TRIAL_VERSION":

    //Parent Refers Child Object 2
    objVS = new ClsTrialVersion(); //Dynamic or Runtime Binding
    break;
}

//Method fnStartScan() is the Version of ClsTrialVersion()
objVS.fnStartScan();

}
}

```

Scan for Internet Security
Detect Virus & Malware

アップキャストिंग

```

objVS = new ClsFreeVersion();
objVS = new ClsPaidVersion();
objVS = new ClsTrialVersion()

```

オンラインでメソッドディスパッチをむ <https://riptutorial.com/ja/java/topic/9204/メソッドディスパッチ>

147: タイプ

き

Java Atomic Typesは、ロックにすることなくスレッドセーフでアトミックなをするシンプルなです。ロックがのボトルネックとなるや、デッドロックやライブロックのがあるにすることをしてい

パラメーター

パラメータ	
セット	フィールドののセット
する	フィールドのみり
レイジーセット	これは、フィールドのオペレーションです
compareAndSet	がexceedならば、それをしいにる
getAndSet	のをしてする

には、のみりまたはきみとCASのみわせがあります。これをするもいは、ソースコードをることです。 [AtomicInteger](#)、 [Unsafe.getAndSet](#)

Examples

タイプの

なマルチスレッドコードでは、[をすることは](#)されます。しかし、[をすることは](#)にし、コードベースがになるにつれ、[デッドロック](#)、[、またはLivelock](#)でわるがくなります。

よりなのは、メソッドまたはコードブロックをするオーバーヘッドなしに々のにスレッドセーフなでアクセスできるため、Atomic Variablesをするがれています。

AtomicIntegerの

```
AtomicInteger aInt = new AtomicInteger() // Create with default value 0
AtomicInteger aInt = new AtomicInteger(1) // Create with initial value 1
```

のインスタンスのもです。

```
AtomicIntegerArray aIntArray = new AtomicIntegerArray(10) // Create array of specific length
AtomicIntegerArray aIntArray = new AtomicIntegerArray(new int[] {1, 2, 3}) // Initialize array
with another array
```

のタイプについてもです。

`float`と`double`がないというながあります。これらは、してシミュレートすることができ
`Float.floatToIntBits(float)`と`Float.intBitsToFloat(int)``float`など`Double.doubleToLongBits(double)`
と`Double.longBitsToDouble(long)`ダブルスのために。

`sun.misc.Unsafe`をするは、`sun.misc.Unsafe`アトミックをして、をアトミックとしてできます。す
べてのプリミティブは、このようにするために、`int`または`long`でまたはエンコードするがありま
す。については、[sun.misc.Unsafe](#)をしてください。

タイプのつけ

マルチスレッドアプリケーションをするなは、Javaのみみのおよびロックプリミティブをするこ
とです。たとえば、`synchronized`キーワードです。のは、`synchronized`をしてカウントをするをし
ています。

```
public class Counters {
    private final int[] counters;

    public Counters(int nosCounters) {
        counters = new int[nosCounters];
    }

    /**
     * Increments the integer at the given index
     */
    public synchronized void count(int number) {
        if (number >= 0 && number < counters.length) {
            counters[number]++;
        }
    }

    /**
     * Obtains the current count of the number at the given index,
     * or if there is no number at that index, returns 0.
     */
    public synchronized int getCount(int number) {
        return (number >= 0 && number < counters.length) ? counters[number] : 0;
    }
}
```

このはしくします。ただし、のスレッドが`Counters`オブジェクトでにのびしをとっているは、が
ボトルネックになりやすくなります。には

1. `synchronized`メソッドびしは、のスレッドが`Counters`インスタンスのロックをしてします。
2. スレッドは、`number`をしているにロックをし、カウンタをします。
3. に、ロックをし、のスレッドにアクセスをします。

1つのスレッドが別のスレッドをしているにロックをしようとする、ロックがされるまで、スレッドはステップ1でブロックされます。別のスレッドがしている、それらのスレッドの1つがし、別のスレッドはききブロックされます。

これはいくつかのにつながります

- ロックにのがあるつまり、別のスレッドがしようとする、別のスレッドはブロックされるがあります。
- スレッドがロックをとってブロックされると、オペレーティングシステムは、別のスレッドにスイッチのをみます。このコンテキスト例えば、プロセッサにきなパフォーマンスのをえます。
- もしロックに別のスレッドがブロックされている、それらのスレッドのいずれかが「」にわれるつまり、スレッドのがされることはされません。これはスレッドのにつながります。

どのようにをするのですか

`AtomicInteger` カウンタをしてのをきしてみましよう。

```
public class Counters {
    private final AtomicInteger[] counters;

    public Counters(int nosCounters) {
        counters = new AtomicInteger[nosCounters];
        for (int i = 0; i < nosCounters; i++) {
            counters[i] = new AtomicInteger();
        }
    }

    /**
     * Increments the integer at the given index
     */
    public void count(int number) {
        if (number >= 0 && number < counters.length) {
            counters[number].incrementAndGet();
        }
    }

    /**
     * Obtains the current count of the object at the given index,
     * or if there is no number at that index, returns 0.
     */
    public int getCount(int number) {
        return (number >= 0 && number < counters.length) ?
            counters[number].get() : 0;
    }
}
```

`int[]` を `AtomicInteger[]` にきえ、それをのインスタンスでしました。また、 `int` ののわりに `incrementAndGet()` と `get()` びしをしました。

しかし、もなことは、ロックがもはやでないため、 `synchronized` キーワードをできることです。

これは、`incrementAndGet()` および `get()` がアトミックでスレッドセーフであるためにします。このでは、それは

- のカウンタは、の「インクリメント」などまたは「」のいずれかでのみです。
- オペレーションが_Tにするとすると、スレッドは_Tの「」をすることができない。

さらに、2つのスレッドがに`AtomicInteger`インスタンスをにしようとしているも、オペレーションのは、されたインスタンスにしてに1つのだけがにわれるようにします。これはロックなしでわれるため、パフォーマンスがすることがあります。

タイプはどのようにしますか

アトミックタイプは、、ターゲットマシンのセットのなハードウェアにします。例えば、Intelベースのセットは、のメモリをにするCAS およびスワップをする。

これらのレベルのは、それぞれの`AtomicXxx`クラスのAPIでレベルのをするためにされます。たとえば、やはりCのようなコードで

```
private volatile num;

int increment() {
    while (TRUE) {
        int old = num;
        int new = old + 1;
        if (old == compare_and_swap(&num, old, new)) {
            return new;
        }
    }
}
```

`AtomicXxxx`にがない、`if`テストはし、ループはただちにします。がある、`if`はスレッドの1つをくすべてのスレッドでし、ループのサイクルがないはループで「」します。には、スピンのいにくくなりますにレベルのをいて、はクラスよりもれていますが、CASがした、ではがえるだけです 1。

ちなみに、CASは、、JVMがのロックをするためにします。ロックがロックされていないことがJVMによってかると、CASをしてロックをしようとします。CASがすれば、なスレッドのスケジューリング、コンテキストのりえなどをうはありません。されるテクニックのについては、「[HotSpotのバイアスされたロック](#)」をしてください。

オンラインでタイプをむ <https://riptutorial.com/ja/java/topic/5963/タイプ>

148: データ

Examples

のインスタンス

```
Object obj = new Object(); // Note the 'new' keyword
```

- `Object` はです。
- `obj` は、しいをするです。
- `Object()` は、 `Object` コンストラクタへのびしです。

がこるのですか

- メモリのがオブジェクトにりてられます。
- コンストラクタ `Object()` は、そのメモリをするためにびされます。
- メモリアドレスは `obj` にされ、しくされたオブジェクトをします。

これはプリミティブとはなりません。

```
int i = 10;
```

の₁₀が_iされている。

はしてこります.オペレーター

```
Object obj = new Object();  
String text = obj.toString(); // 'obj' is dereferenced.
```

は、にされているメモリアドレス、のオブジェクトがするメモリのにいます。オブジェクトがつかった、されたメソッドがびされますこのは `toString`。

のが `null`、により `NullPointerException` がします。

```
Object obj = null;  
obj.toString(); // Throws a NullPointerException when this statement is executed.
```

`null` すなわち、メモリアドレスはどこにもつながりません、がないことをします。したがって、されたメソッドをびすことができるオブジェクトはありません。

オンラインでデータをむ <https://riptutorial.com/ja/java/topic/1046/データ>

149:

Examples

なる

`java.lang.ref` パッケージは、ガベージコレクタとのされたをサポートするオブジェクトクラスをします。

Javaには4つのななるがあります。らです

- い
- い
- ソフトリファレンス
- ファントムリファレンス

1.い

これは、オブジェクトをするのです。

```
MyObject myObject = new MyObject();
```

ホルダーはされたオブジェクトへのいをしています。このが `MyObject` このをしているり、 `MyObject` インスタンスはガベージコレクタによってされません。

2.い

オブジェクトをくしたくないや、できるだけオブジェクトにりてられたメモリをクリア/フリーにするがあるは、そうするがあります。

```
WeakReference myObjectRef = new WeakReference(MyObject);
```

に、とは、オブジェクトをにメモリにすようくくないです。いは、ガベージコレクタのをしてをできるため、ではありません。

したオブジェクトがなは、 `.get()` メソッドをして `.get()`

```
myObjectRef.get();
```

のコードはこれをします

```
WeakReference myObjectRef = new WeakReference(MyObject);
System.out.println(myObjectRef.get()); // This will print the object reference address
System.gc();
System.out.println(myObjectRef.get()); // This will print 'null' if the GC cleaned up the
object
```

3. ソフトリファレンス

ソフトリファレンスはリファレンスよりもわずかにです。のように、ソフトオブジェクトをすることができます。

```
SoftReference myObjectRef = new SoftReference(MyObject);
```

らはいよりもくメモリにすることができます。なメモリ/リソースがあれば、ガベージコレクタはいとしてにソフトリファレンスをクリーンアップしません。

ソフトリファレンスはキャッシュにするとです。ソフトリファレンスオブジェクトをキャッシュとしてし、メモリがなくなるまですることができます。メモリがなリソースをできない、ガベージコレクタはソフトリファレンスをします。

```
SoftReference myObjectRef = new SoftReference(MyObject);
System.out.println(myObjectRef.get()); // This will print the reference address of the Object
System.gc();
System.out.println(myObjectRef.get()); // This may or may not print the reference address of
the Object
```

4. ファントム・リファレンス

これはもいです。Phantom Referenceをしてオブジェクトをした、`get()` メソッドはにnullをします。

こののは、「コレクタがそのをできるとしたにエンキューされるファントムオブジェクト」です。ファントムは、プリコーミングクリーンアップアクションのスケジューリングにもにされます。Javaのファイナライズのみ"- [オントロジーのPhantom Reference Javadoc](#)より。

のように、Phantom Referenceのオブジェクトをできます。

```
PhantomReference myObjectRef = new PhantomReference(MyObject);
```

オンラインでをむ <https://riptutorial.com/ja/java/topic/4017/>

150: クラスのメンバーへのアクセスをする

- パブリックタイプ[=];
- プライベートタイプ[=];
- された[=];
- タイプ[=];
- パブリッククラス{
- クラス{

Javaチュートリアルから

アクセス・レベルは、のクラスがのフィールドをできるか、のメソッドをびすことができるかどうかをします。アクセスには2つのレベルがあります。

- トップレベル - `public`、またはパッケージ - プライベート なし。
- メンバレベル - `public`、`private`、`protected`、`package-private` なし。

クラスは、`public`ですることができます。この、そのクラスはすべてのクラスすべてでです。クラスにがないデフォルトは`package-private`ともばれます、それはそのパッケージでしかることができません。

メンバレベルでは、トップレベルクラスのとに`public`をすることも、じをしない `package-private` こともできます。メンバーには、さらに2つのアクセス、`private`と`protected`ます。`private`は、そのメンバーがのクラスでのみアクセスであることをします。`protected`は、そのメンバが `package-private`のように そのパッケージだけでアクセスでき、さらにそのクラスのサブクラス によってのパッケージにアクセスできることをします。

のは、でされているメンバーへのアクセスをしています。

アクセスレベル

モディファイア	クラス	パッケージ	サブクラス	
<code>public</code>	Y	Y	Y	Y
<code>protected</code>	Y	Y	Y	N
なし	Y	Y	N	N
<code>private</code>	Y	N	N	N

Examples

インタフェースメンバ

```

public interface MyInterface {
    public void foo();
    int bar();

    public String TEXT = "Hello";
    int ANSWER = 42;

    public class X {
    }

    class Y {
    }
}

```

`public` キーワードがされていても、インターフェースメンバーはにされます。したがって、`foo()`、`bar()`、`TEXT`、`ANSWER`、`X`、および`Y`がされています。 `MyInterface` はされているため、どこからでもアクセスできますが、`MyInterface` パッケージのがあった、そのメンバーはじパッケージからのみアクセスできます。

クラス、パッケージ、およびサブクラスにされます。

Testクラスのを試みましょう。

```

public class Test{
    public int number = 2;

    public Test(){

    }
}

```

では、クラスのインスタンスをしようします。このでは、`public` いるため `number` アクセスできます。

```

public class Other{

    public static void main(String[] args){
        Test t = new Test();
        System.out.println(t.number);
    }

}

```

な

`private` は、そのクラスによってのみにアクセスできるようにします。らはしばしば `public` ゲッターやセッターとにされます。

```

class SomeClass {
    private int variable;

    public int getVariable() {
        return variable;
    }
}

```

```

    }

    public void setVariable(int variable) {
        this.variable = variable;
    }
}

public class SomeOtherClass {
    public static void main(String[] args) {
        SomeClass sc = new SomeClass();

        // These statement won't compile because SomeClass#variable is private:
        sc.variable = 7;
        System.out.println(sc.variable);

        // Instead, you should use the public getter and setter:
        sc.setVariable(7);
        System.out.println(sc.getVariable());
    }
}

```

パッケージの

をしないのデフォルトは、**package visibility**です。Javaドキュメンテーションから、「**[package visibility]**は、クラスとしパッケージのクラスそのになくがメンバにアクセスできるかどうかをします。 [javax.swing](#)このでは、

```

package javax.swing;
public abstract class JComponent extends Container ... {
    ...
    static boolean DEBUG_GRAPHICS_LOADED;
    ...
}

```

`DebugGraphics`はしパッケージにまれているため、`DEBUG_GRAPHICS_LOADED`にアクセスできます。

```

package javax.swing;
public class DebugGraphics extends Graphics {
    ...
    static {
        JComponent.DEBUG_GRAPHICS_LOADED = true;
    }
    ...
}

```

このでは、このトピックのについてします。

された

されたのは、このメンバーがそのサブクラスとともにパッケージでであることをします。

として

```
package com.stackexchange.docs;
public class MyClass{
    protected int variable; //This is the variable that we are trying to access
    public MyClass(){
        variable = 2;
    };
}
```

に、このクラスをし、 `protected` メンバーの1つにアクセスしようとしています。

```
package some.other.pack;
import com.stackexchange.docs.MyClass;
public class SubClass extends MyClass{
    public SubClass(){
        super();
        System.out.println(super.variable);
    }
}
```

`protected` メンバーにしパッケージからアクセスしているは、そのメンバーをせずにアクセスすることもできます。

これはクラスのメンバーでのみし、クラスではないことにしてください。

クラスメンバアクセスの

アクセス		
プライベート	クラスのみ	することはできません
パッケージなし	パッケージ	パッケージのサブクラスのに
された	パッケージ	サブクラスで
パブリック	どこにでも	サブクラスで

メソッドやにしてパッケージのサブクラスからアクセスできるようにする `private` `protected` へのキーワードがありましたが、そのパッケージのクラスにしてにしました。ただし、これは [Java 1.0](#) のリリースではされませんでした。

オンラインでクラスのメンバーへのアクセスをするをむ <https://riptutorial.com/ja/java/topic/134/-クラスのメンバーへのアクセスをする->

151:

き

[java.util.Map](#) [インタフェース](#)は、キーとそのもののマッピングをします。マップにキーをめることはできません。キーはで1つのにマップできます。

`Map`はインターフェイスなので、インターフェイスをするためには、そのインターフェイスのなをインスタンスするがあります。いくつかの`Map`があり、に `java.util.HashMap`と `java.util.TreeMap`がされます

[マップ](#)は、キーごとにするをつキーをするオブジェクトです。キーとそのは、キー/のペアまたはエントリと呼ばれることもあります。、マップにはのがあります。

- データはキー/のペアでマップにされます。
- マップには、のキーのエントリが1つしかまれていないがあります。マップにのキーをつエントリがまれていて、じキーをつ2のエントリをしようすると、2のエントリがのエントリにきわります。つまり、これはキーにけられたをします。
- マップは、キーがマップにするかどうかをテストし、キーにけられたをフェッチし、キーとのペアをするためのをします。

もにされるマップは[HashMap](#)です。やのキーでうまくします。

`HashMap`などのプレーンマップはけされていません。キーとのペアをすることで、々のエントリをのことができます。されたでマップエントリをするがあるは、をするがあります。

- [TreeMap](#)のような[ソートされたマップ](#)は、でキーをしますまたは、[Comparator](#)をすることでできるで。たとえば、をキーとしてソートされたマップは、にそのをすることがされます。
- [LinkedHashMap](#)は、マップにされたとしで、またはアクセスされたでエントリをすることをします。

Examples

をする

1.

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "First element.");
System.out.println(map.get(1));
```

First element.

2. オーバーライド

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "First element.");
map.put(1, "New element.");
System.out.println(map.get(1));
```

New element.

HashMapとしてします。 Mapインタフェースをするのにもできます。

のをする

V put(K key,V value)をうことができます

このマップのされたキーにされたをけますオプション。マップにキーのマッピングが
まれていた、いはされたにきえられます。

```
String currentVal;
Map<Integer, String> map = new TreeMap<>();
currentVal = map.put(1, "First element.");
System.out.println(currentVal); // Will print null
currentVal = map.put(2, "Second element.");
System.out.println(currentVal); // Will print null yet again
currentVal = map.put(2, "This will replace 'Second element'");
System.out.println(currentVal); // will print Second element.
System.out.println(map.size()); // Will print 2 as key having
// value 2 was replaced.

Map<Integer, String> map2 = new HashMap<>();
map2.put(2, "Element 2");
map2.put(3, "Element 3");

map.putAll(map2);

System.out.println(map.size());
```

3

くのをするには、のようなクラスをできます。

```
Map<Integer, String> map = new HashMap<>() {{
    // This is now an anonymous inner class with an unnamed instance constructor
    put(5, "high");
    put(4, "low");
    put(1, "too slow");
}};
```

のクラスのはずしもではなく、メモリリークをくがあるので、であれば、わりにイニシャライザ
ブロックをしてください。

```
static Map<Integer, String> map = new HashMap<>();

static {
```

```
// Now no inner classes are created so we can avoid memory leaks
put(5, "high");
put(4, "low");
put(1, "too slow");
}
```

のは、マップをにします。また、でないコンテキストでも、すべてのなものをりくことででき
static。

これにえて、ほとんどのでputAllサポートされています。これは、あるマップのすべてのエント
リをののようにのマップにできます。

```
another.putAll(one);
```

Java 8からのマップのデフォルトメソッドの

マップインタフェースのJava 8でされたデフォルトメソッドの

1. getOrDefaultの

キーにマップされたをします。キーがしないは、デフォルトをします。

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "First element");
map.get(1); // => First element
map.get(2); // => null
map.getOrDefault(2, "Default element"); // => Default element
```

2. forEachをする

マップエントリで'アクション'でされたをできます

```
Map<Integer, String> map = new HashMap<Integer, String>();
map.put(1, "one");
map.put(2, "two");
map.put(3, "three");
map.forEach((key, value) -> System.out.println("Key: "+key+ " :: Value: "+value));

// Key: 1 :: Value: one
// Key: 2 :: Value: two
// Key: 3 :: Value: three
```

3. replaceAllをする

キーがあるのみしいできえられます

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("john", 20);
map.put("paul", 30);
map.put("peter", 40);
map.replaceAll((key, value) -> value+10); // {john=30, paul=40, peter=50}
```

4. putIfAbsentの

キーがないか、またはnullにマップされているは、キーとのペアがマップにされます

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("john", 20);
map.put("paul", 30);
map.put("peter", 40);
map.putIfAbsent("kelly", 50);    //{john=20, paul=30, peter=40, kelly=50}
```

5. の

されたにけられているにのみキーをします。

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("john", 20);
map.put("paul", 30);
map.put("peter", 40);
map.remove("peter", 40); //{john=30, paul=40}
```

6. きえをする

キーがする、はしいにきえられます。キーがない、もしません。

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("john", 20);
map.put("paul", 30);
map.put("peter", 40);
map.replace("peter", 50); //{john=20, paul=30, peter=50}
map.replace("jack", 60); //{john=20, paul=30, peter=50}
```

7. computeIfAbsentの

このメソッドは、マップにエントリをします。キーはでされ、はマッピングのです

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("john", 20);
map.put("paul", 30);
map.put("peter", 40);
map.computeIfAbsent("kelly", k->map.get("john")+10); //{john=20, paul=30, peter=40, kelly=30}
map.computeIfAbsent("peter", k->map.get("john")+10); //{john=20, paul=30, peter=40, kelly=30} //peter already present
```

8. computeIfPresentをう

このメソッドは、エントリをするか、マップののエントリをします。そのキーをつエントリがないはもしません

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("john", 20);
map.put("paul", 30);
```

```
map.put("peter", 40);
map.computeIfPresent("kelly", (k,v)->v+10); //{john=20, paul=30, peter=40} //kelly not
present
map.computeIfPresent("peter", (k,v)->v+10); //{john=20, paul=30, peter=50} // peter
present, so increase the value
```

9. をう

このメソッドは、キーのをしくされたでええます

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("john", 20);
map.put("paul", 30);
map.put("peter", 40);
map.compute("peter", (k,v)->v+50); //{john=20, paul=30, peter=90} //Increase the value
```

10. マージの

キーがしないはキーをにきえます。キーがするは、しくされたにきえます。キーがマップからされます。しいがnullのはマップにされます。

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("john", 20);
map.put("paul", 30);
map.put("peter", 40);

//Adds the key-value pair to the map, if key is not present or value for the key is null
map.merge("kelly", 50 , (k,v)->map.get("john")+10); // {john=20, paul=30, peter=40,
kelly=50}

//Replaces the value with the newly computed value, if the key is present
map.merge("peter", 50 , (k,v)->map.get("john")+10); //{john=20, paul=30, peter=30,
kelly=50}

//Key is removed from the map , if new value computed is null
map.merge("peter", 30 , (k,v)->map.get("nancy")); //{john=20, paul=30, kelly=50}
```

をクリアする

```
Map<Integer, String> map = new HashMap<>();

map.put(1, "First element.");
map.put(2, "Second element.");
map.put(3, "Third element.");

map.clear();

System.out.println(map.size()); // => 0
```

マップのをする

マップでは、マップのキー、、またはキーとのペアにコレクションとしてアクセスできるメソッドがされています。これらのコレクションをすることができます。たとえば、のマップをええます

。

```
Map<String, Integer> repMap = new HashMap<>();  
repMap.put("Jon Skeet", 927_654);  
repMap.put("BalusC", 708_826);  
repMap.put("Darin Dimitrov", 715_567);
```

マップキーをする

```
for (String key : repMap.keySet()) {  
    System.out.println(key);  
}
```

ダーリン・ディミトロフ
ジョン・スケート
BalusC

`keySet()` はマップのキーを `Set` としてします。キーはするをむことができないため、`Set` がされます。セットをすると、にキーがされます。 `HashMaps` はけされていないので、このではキーはのされます。

マップをする

```
for (Integer value : repMap.values()) {  
    System.out.println(value);  
}
```

715567
927654
708826

`values()` は、マップのを `Collection` としてします。コレクションをすると、にがされます。ここでも、はのされます。

キーとをにりす

```
for (Map.Entry<String, Integer> entry : repMap.entrySet()) {  
    System.out.printf("%s = %d\n", entry.getKey(), entry.getValue());  
}
```

ダーリン・ディミトロフ = 715567
ジョン・スケート = 927654
BalusC = 708826

`entrySet()` は、 `Map.Entry` オブジェクトのコレクションをします。 `Map.Entry` はエントリのキーとにアクセスします。

マップのマージ、

`putAll`をして、あるマップのすべてのメンバーをのマップにします。マップにすでにするキーには、するがきされます。

```
Map<String, Integer> numbers = new HashMap<>();
numbers.put("One", 1)
numbers.put("Three", 3)
Map<String, Integer> other_numbers = new HashMap<>();
other_numbers.put("Two", 2)
other_numbers.put("Three", 4)

numbers.putAll(other_numbers)
```

これにより、のようなマッピングが`numbers`られ`numbers`。

```
"One" -> 1
"Two" -> 2
"Three" -> 4 //old value 3 was overwritten by new value 4
```

をきするのではなく、をするは、Java 8でされた`Map.merge`をできます。これは、ユーザーの `BiFunction` をしてキーのをマージします。 `merge` は々のキーとですするため、ループまたは `Map.forEach` をするがあります。ここでは、するキーのをします。

```
for (Map.Entry<String, Integer> e : other_numbers.entrySet())
    numbers.merge(e.getKey(), e.getValue(), Integer::sum);
//or instead of the above loop
other_numbers.forEach((k, v) -> numbers.merge(k, v, Integer::sum));
```

をするは、キーがしないようにするには、 `AssertionError` をスローするマージをします。

```
mapA.forEach((k, v) ->
    mapB.merge(k, v, (v1, v2) ->
        {throw new AssertionError("duplicate values for key: "+k);}));
```

Map <X、 Y>と Map <Y、 Z>をして Map <X、 Z>をします。

2つのマッピングをするは、のようになことができます

```
Map<String, Integer> map1 = new HashMap<String, Integer>();
map1.put("key1", 1);
map1.put("key2", 2);
map1.put("key3", 3);

Map<Integer, Double> map2 = new HashMap<Integer, Double>();
map2.put(1, 1.0);
map2.put(2, 2.0);
map2.put(3, 3.0);

Map<String, Double> map3 = new new HashMap<String, Double>();
map1.forEach((key,value)->map3.put(key,map2.get(value)));
```

これにより、のマッピング

```
"key1" -> 1.0
"key2" -> 2.0
"key3" -> 3.0
```

キーがするかどうかをする

```
Map<String, String> num = new HashMap<>();
num.put("one", "first");

if (num.containsKey("one")) {
    System.out.println(num.get("one")); // => first
}
```

マップには**NULL**をめることができます

マップの、「キーをむ」と「をつ」をしないようにしなければなりません。たとえば、`HashMap`は `null`がまれているがあります。これは、のことがになであることをします。

```
Map<String, String> map = new HashMap<>();
map.put("one", null);
if (map.containsKey("one")) {
    System.out.println("This prints !"); // This line is reached
}
if (map.get("one") != null) {
    System.out.println("This is never reached !"); // This line is never reached
}
```

よりには、`map.containsKey(key) <=> map.get(key) != null` というのはありません `map.containsKey(key) <=> map.get(key) != null`

なエントリの

このセクションでは、`Map<Integer, Integer>`のエントリをりしし、`Integer`のをする10のののコードとベンチマークをします。すべてののはアルゴリズムのさが $\Theta(n)$ ですが、のでどのがよりであるかについてののをするのにちます。

1. `Map.Entry`をした `Iterator`をした

```
Iterator<Map.Entry<Integer, Integer>> it = map.entrySet().iterator();
while (it.hasNext()) {
    Map.Entry<Integer, Integer> pair = it.next();
    sum += pair.getKey() + pair.getValue();
}
```

2. `for Map.Entry`を `for`

```
for (Map.Entry<Integer, Integer> pair : map.entrySet()) {
    sum += pair.getKey() + pair.getValue();
}
```

3. `Map.forEach` Java 8+をした

```
map.forEach((k, v) -> sum[0] += k + v);
```

4. の `for Map.keySet` をった

```
for (Integer key : map.keySet()) {
    sum += key + map.get(key);
}
```

5. `Iterator` をした `Map.keySet` をした

```
Iterator<Integer> it = map.keySet().iterator();
while (it.hasNext()) {
    Integer key = it.next();
    sum += key + map.get(key);
}
```

6. `for Iterator` と `Map.Entry` をった

```
for (Iterator<Map.Entry<Integer, Integer>> entries =
    map.entrySet().iterator(); entries.hasNext(); ) {
    Map.Entry<Integer, Integer> entry = entries.next();
    sum += entry.getKey() + entry.getValue();
}
```

7. `Stream.forEach` Java 8をした

```
map.entrySet().stream().forEach(e -> sum += e.getKey() + e.getValue());
```

8. `Stream.forEach` と `Stream.parallel` Java 8+をした

```
map.entrySet()
    .stream()
    .parallel()
    .forEach(e -> sum += e.getKey() + e.getValue());
```

9. Apacheコレクションからの `IterableMap` をした

```
MapIterator<Integer, Integer> mit = iterableMap.mapIterator();
while (mit.hasNext()) {
    sum += mit.next() + it.getValue();
}
```

10. Eclipseコレクションから `MutableMap` をした


```
mutableMap.forEachKeyValue((key, value) -> {
    sum += key + value;
});
```

パフォーマンステスト [Github](#)でなコード

テストWindows 8.1 64ビット、Intel i7-4790 3.60GHz、16GB

1. 10の100の308±21ns / op

Benchmark	Score	Error	Units
test3_UsingForEachAndJava8	308 ±	21	ns/op
test10_UsingEclipseMutableMap	309 ±	9	ns/op
test1_UsingWhileAndMapEntry	380 ±	14	ns/op
test6_UsingForAndIterator	387 ±	16	ns/op
test2_UsingForEachAndMapEntry	391 ±	23	ns/op
test7_UsingJava8StreamAPI	510 ±	14	ns/op
test9_UsingApacheIterableMap	524 ±	8	ns/op
test4_UsingKeySetAndForEach	816 ±	26	ns/op
test5_UsingKeySetAndIterator	863 ±	25	ns/op
test8_UsingJava8StreamAPIParallel	5552 ±	185	ns/op

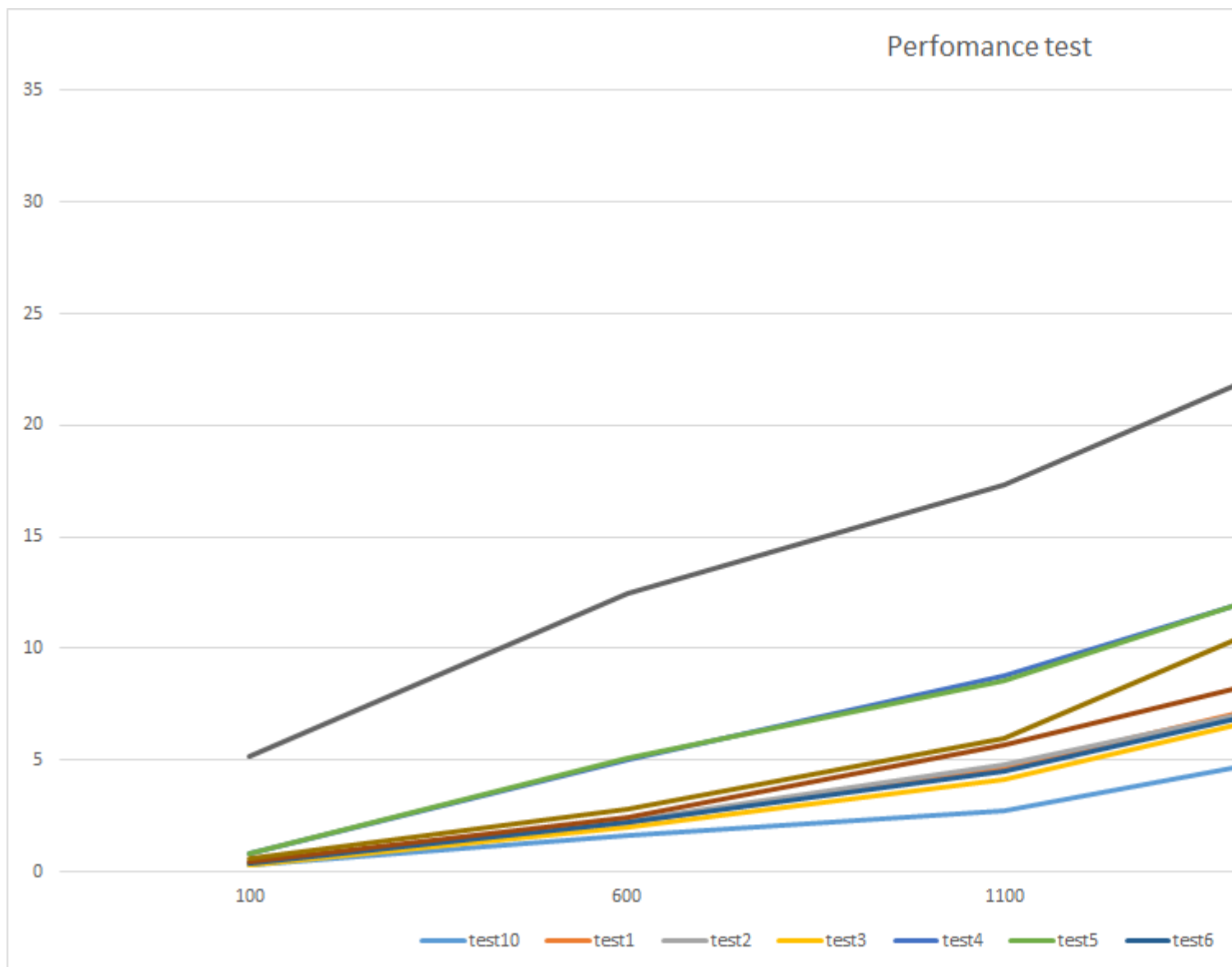
2. 10の10000の37.606±0.790µs/ op

Benchmark	Score	Error	Units
test10_UsingEclipseMutableMap	37606 ±	790	ns/op
test3_UsingForEachAndJava8	50368 ±	887	ns/op
test6_UsingForAndIterator	50332 ±	507	ns/op
test2_UsingForEachAndMapEntry	51406 ±	1032	ns/op
test1_UsingWhileAndMapEntry	52538 ±	2431	ns/op
test7_UsingJava8StreamAPI	54464 ±	712	ns/op
test4_UsingKeySetAndForEach	79016 ±	25345	ns/op
test5_UsingKeySetAndIterator	91105 ±	10220	ns/op
test8_UsingJava8StreamAPIParallel	112511 ±	365	ns/op
test9_UsingApacheIterableMap	125714 ±	1935	ns/op

3. 10の100000の1184.767±332.968µs/ op

Benchmark	Score	Error	Units
test1_UsingWhileAndMapEntry	1184.767 ±	332.968	µs/op
test10_UsingEclipseMutableMap	1191.735 ±	304.273	µs/op
test2_UsingForEachAndMapEntry	1205.815 ±	366.043	µs/op
test6_UsingForAndIterator	1206.873 ±	367.272	µs/op
test8_UsingJava8StreamAPIParallel	1485.895 ±	233.143	µs/op
test5_UsingKeySetAndIterator	1540.281 ±	357.497	µs/op
test4_UsingKeySetAndForEach	1593.342 ±	294.417	µs/op
test3_UsingForEachAndJava8	1666.296 ±	126.443	µs/op
test7_UsingJava8StreamAPI	1706.676 ±	436.867	µs/op
test9_UsingApacheIterableMap	3289.866 ±	1445.564	µs/op

4. マップサイズにするの



x: Size of Map
f(x): Benchmark Score (μs/op)

		100	600	1100	1600	2100
Tests f(x)	10	0.333	1.631	2.752	5.937	8.024
	3	0.309	1.971	4.147	8.147	10.473
	6	0.372	2.190	4.470	8.322	10.531
	1	0.405	2.237	4.616	8.645	10.707
	2	0.376	2.267	4.809	8.403	10.910
	7	0.473	2.448	5.668	9.790	12.125
	9	0.565	2.830	5.952	13.22	16.965
	4	0.808	5.012	8.813	13.939	17.407
	5	0.81	5.104	8.533	14.064	17.422
	8	5.173	12.499	17.351	24.671	30.403

カスタムオブジェクトをキーとしてする

のオブジェクトをキーとしてするに、オブジェクトのhashCodeおよびequalsメソッドをオーバーライドする必要があります。

な、あなたはのようなものを持っています

```
class MyKey {
    private String name;
    MyKey(String name) {
        this.name = name;
    }

    @Override
    public boolean equals(Object obj) {
        if(obj instanceof MyKey) {
            return this.name.equals(((MyKey)obj).name);
        }
        return false;
    }

    @Override
    public int hashCode() {
        return this.name.hashCode();
    }
}
```

`hashCode`は、キーがするハッシュバケットをし、`equals`はそのハッシュバケットのオブジェクトをします。

これらのメソッドがなければ、オブジェクトのはのにされ、じオブジェクトをしなければなりません。

HashMapの

HashMapは、Key-ValueペアにデータをするデータをするMapインタフェースのです。

1. HashMapの

```
Map<KeyType, ValueType> myMap = new HashMap<KeyType, ValueType>();
```

`KeyType`と`ValueType`は、`String`、`Integer`、`Float`、`Employee`、`Student`などのカスタムクラスのよ
うなJavaのなでなければなりません。

```
Map<String,Integer> myMap = new HashMap<String,Integer>();
```

2. をHashMapにれます。

HashMapにをれるには、`Key`と`Value`をパラメータとしてしてHashMapオブジェクトに`put`メソッ
ドをびさなければなりません。

```
myMap.put("key1", 1);
myMap.put("key2", 2);
```

MapにすでにするKeyをして`put`メソッドをびすと、メソッドはそのをオーバーライドしていをし
ます。

3. HashMapからををする。

HashMapからを`get`するには、`Key`をパラメータとしてすることによって`get`メソッドをびすがあります。

```
myMap.get("key1"); //return 1 (class Integer)
```

HashMapにしないキーをすと、このメソッドは`null`をし`null`

4. キーがマップにあるかどうかをします。

```
myMap.containsKey(varKey);
```

5. かにあるかどうかをします。

```
myMap.containsValue(varValue);
```

のメソッドは、`key`、`value`がMapにするかどうかにかかわらずまたはの`boolean`をします。

マップのと

き

`Maps`はキーとのペアがされます。キーにはするがあります。のキーがえられた、マップはけられたをににルックアップできます。

アソシエートともばれる`Maps`は、キーとのでデータをするオブジェクトです。Javaでは、マップはコレクションインターフェイスではないMapインターフェイスをしてされます。

- 1 -

```
/*J2SE < 5.0*/
Map map = new HashMap();
map.put("name", "A");
map.put("address", "Malviya-Nagar");
map.put("city", "Jaipur");
System.out.println(map);
```

- 2 -

```
/*J2SE 5.0+ style (use of generics):*/
Map<String, Object> map = new HashMap<>();
map.put("name", "A");
map.put("address", "Malviya-Nagar");
map.put("city", "Jaipur");
System.out.println(map);
```

-

3 -

```
Map<String, Object> map = new HashMap<String, Object>(){  
    put("name", "A");  
    put("address", "Malviya-Nagar");  
    put("city", "Jaipur");  
};  
System.out.println(map);
```

• 4 -

```
Map<String, Object> map = new TreeMap<String, Object>();  
map.put("name", "A");  
map.put("address", "Malviya-Nagar");  
map.put("city", "Jaipur");  
System.out.println(map);
```

• 5 -

```
//Java 8  
final Map<String, String> map =  
    Arrays.stream(new String[][] {  
        { "name", "A" },  
        { "address", "Malviya-Nagar" },  
        { "city", "jaipur" },  
    }).collect(Collectors.toMap(m -> m[0], m -> m[1]));  
System.out.println(map);
```

• 6 -

```
//This way for initial a map in outside the function  
final static Map<String, String> map;  
static  
{  
    map = new HashMap<String, String>();  
    map.put("a", "b");  
    map.put("c", "d");  
}
```

• Way 7 - のキーマップの。

```
//Immutable single key-value map  
Map<String, String> singletonMap = Collections.singletonMap("key", "value");
```

このをすることはであることにしてください。

マップをしようとすると、`UnsupportedOperationException`がスローされます。

```
//Immutable single key-value pair  
Map<String, String> singletonMap = Collections.singletonMap("key", "value");  
singletonMap.put("newKey", "newValue"); //will throw UnsupportedOperationException  
singletonMap.putAll(new HashMap<>()); //will throw UnsupportedOperationException  
singletonMap.remove("key"); //will throw UnsupportedOperationException
```

```
singletonMap.replace("key", "value", "newValue"); //will throw  
UnsupportedOperationException  
//and etc
```

オンラインでをむ <https://riptutorial.com/ja/java/topic/105/>

152: なコントロール

すべてののは、にしないり、 **ブロック**をします。これらは `{ }` されます。

これは、をとしない **のステートメント**とはなりますが、のステートメントののだけがされるといいうしいもあります。

したがって、をわないでこれらのをくことはにです。にくは1つだけですが、バグのいやれたコードにつながるがあるため、くおめします。

```
// valid, but discouraged
Scanner scan = new Scanner(System.in);
int val = scan.nextInt();
if(val % 2 == 0)
    System.out.println("Val was even!");

// invalid; will not compile
// note the misleading indentation here
for(int i = 0; i < 10; i++)
    System.out.println(i);
    System.out.println("i is currently: " + i);
```

Examples

If / Else If / Else コントロール

```
if (i < 2) {
    System.out.println("i is less than 2");
} else if (i > 2) {
    System.out.println("i is more than 2");
} else {
    System.out.println("i is not less than 2, and not more than 2");
}
```

`if` ブロックは、`i` が1のときにのみされます。

`else if` は、そののすべてのの `else if` コンストラクト、および `parent if` コンストラクトが `false` テストされている `if` のみチェックされ `false`。このでは、`else if` は、`i` が2ののみにチェックされます。

そのが `true` であれば、そのブロックがされ、`else` の `else if` および `else` コンストラクトはスキップされます。

どれ `if` とは `else if` のがにテストされた `true`、`else` のブロックはされません。

For Loops

```
for (int i = 0; i < 100; i++) {
    System.out.println(i);
}
```

3つのforループによって、`for (int i = 0; i < 100; i++)` の3つの部分で、`int i = 0`、`i < 100`、およびステートメント `i++`。のは、`{` の中に `{` にかれたかのように1されます。そして、`for` がチェックされていることがあるは、`true` ループのは、それが `for` があるは、される `false` ループがします。ループがくと、ボディがされ、`}` にすると、`for` がびチェックされるにインクリメントステートメントがされます。

ループに1つのステートメントだけがまれている、はオプションですセミコロンで1にすることができます。しかし、やバグをけるためにをすることがにされます。

`for` ループコンポーネントはオプションです。ビジネスロジックにこれらのパーツがまれているは、`for` ループからするコンポーネントをできます。

```
int i = obj.getLastestValue(); // i value is fetched from a method

for (; i < 100; i++) { // here initialization is not done
    System.out.println(i);
}
```

`for (;;) { function-body }` は、`while (true)` ループと同じです。

Nested For Loops

ネストされたループと呼ばれるにのループステートメントをつループステートメント。よりくのループをつループのとは、`'nested for loop'` とされます。

```
for(;;){
    //Outer Loop Statements
    for(;;){
        //Inner Loop Statements
    }
    //Outer Loop Statements
}
```

にされたforループは、のをすることができます。

```
for(int i=9;i>0;i--){ //Outer Loop
    System.out.println(i);
    for(int k=i;k>0;k--){ //Inner Loop -1
        System.out.print(" ");
    }
    for(int j=i;j<=9;j++){ //Inner Loop -2
        System.out.print(" "+j);
    }
}
```

Whileループ

```
int i = 0;
```



```
while (i < 100) { // condition gets checked BEFORE the loop body executes
    System.out.println(i);
    i++;
}
```

カッコのが`true`り、`while`ループがされ`true`。これは、メインループがされるにがたされなければならないため、「テストループ」ともされます。

ループに1つのステートメントしかまかれていないは、はオプションですが、いくつかのコーディングスタイルのでは、をとしません。

do ... whileループ

`do...while`ループは、なくとも1はすることがされているで、のループとはなります。はメインループのにされるため、「テストのループ」ともされます。

```
int i = 0;
do {
    i++;
    System.out.println(i);
} while (i < 100); // Condition gets checked AFTER the content of the loop executes.
```

このでは、ループがされたにループがされるため、`100`がされるまでループがされますは`i < 100`で`i <= 100`ありません。

なくとも1つのをすることで、ループのをしてですることができます。

```
String theWord;
Scanner scan = new Scanner(System.in);
do {
    theWord = scan.nextLine();
} while (!theWord.equals("Bird"));

System.out.println(theWord);
```

このコンテキストでは、`theWord`はループのでされますが、なフローについてがされるため、`theWord`はされます。

For Each

Java SE 5

Java 5では、`for-each`ループ`for`ループともされますをできます。

```
List strings = new ArrayList();

strings.add("This");
strings.add("is");
strings.add("a for-each loop");

for (String string : strings) {
```

```
System.out.println(string);
}
```

ループは `Iterable` インタフェースの `iterator()` をするためにでき、`List` や `Set` などの `Collections` クラスが
 含まれます。

ループは、ソース・タイプから `Iterable` のタイプにすることができます。

`Iterable<T>` または `T[]` `for` ループのループは、タイプ `S` があります `Iterable<T>`

- `T` extends `S`
- `T` と `S` がプリミティブであり、キャストなしで使えます
- `S` はプリミティブであり、`T` は `unboxing` に `S` になれます。
- `T` はプリミティブであり、ボックスングによって `S` にすることができます。

```
T elements = ...
for (S s : elements) {
}
```

T	S	コンパイルする
<code>int []</code>	<code>い</code> です	はい
<code>い</code> です[]	<code>int</code>	いいえ
<code>Iterable<Byte></code>	<code>い</code> です	はい
<code>Iterable<String></code>	<code>CharSequence</code>	はい
<code>Iterable<CharSequence></code>		いいえ
<code>int []</code>	<code>い</code> です	いいえ
<code>int []</code>		はい

If / Else

```
int i = 2;
if (i < 2) {
    System.out.println("i is less than 2");
} else {
    System.out.println("i is greater than 2");
}
```

`if` は、条件が真のときにコードを実行します。条件が偽の場合、`{ }` で囲まれた `if` のブロックに属するコードは実行されません。条件が真の場合、`if` ステートメントの範囲になるまで、括弧を閉じます。

`else` ブロックはオプションです。条件が `false` の場合、`if` が `true` の場合、`if` は実行され

ん。この`if`がされるためです。

Ternary Ifもしてください。

スイッチ

`switch`は、Javaのです。い`if - else if - else`チェーンのわりにして、それらをよりみやすくします。ただし、`if`とはなり`if`はできません。をにするがあります。

`switch`には3つのなコンポーネントがあります。

- `case` これは`switch`のとであるかどうかされるです。
- `default` これはオプションのキャッチオールであり、`case`のどれもが`true`され`true case`り`true`。
- `case`ステートメントの。 `break` これは、さらに、ましくないをするためにとされる`case`ステートメントを。

`continue`いて、ステートメントののをきこすステートメントをすることはです。これも

- `break`
- `return`
- `throw`

のでは、な`switch`ステートメントが、`default`をむ4つのなケースでされてい`default`。

```
Scanner scan = new Scanner(System.in);
int i = scan.nextInt();
switch (i) {
    case 0:
        System.out.println("i is zero");
        break;
    case 1:
        System.out.println("i is one");
        break;
    case 2:
        System.out.println("i is two");
        break;
    default:
        System.out.println("i is less than zero or greater than two");
}
```

`break`やもないをうステートメントを`break`ことで、いくつかのにしてされる「フォールスルー」ケースとしてられるものをすることができます。これは、がするをするのにできますが、ほどではありません。

```
Scanner scan = new Scanner(System.in);
int foo = scan.nextInt();
switch(foo) {
    case 1:
        System.out.println("I'm equal or greater than one");
    case 2:
    case 3:
```

```

        System.out.println("I'm one, two, or three");
        break;
    default:
        System.out.println("I'm not either one, two, or three");
    }

```

foo == 1、はのようになります。

```

I'm equal or greater than one
I'm one, two, or three

```

foo == 3、はのようになります。

```

I'm one, two, or three

```

Java SE 5

switchは、enumですることもできます。

```

enum Option {
    BLUE_PILL,
    RED_PILL
}

public void takeOne(Option option) {
    switch(option) {
        case BLUE_PILL:
            System.out.println("Story ends, wake up, believe whatever you want.");
            break;
        case RED_PILL:
            System.out.println("I show you how deep the rabbit hole goes.");
            break;
    }
}

```

Java SE 7

switchは、Stringですることもできます。

```

public void rhymingGame(String phrase) {
    switch (phrase) {
        case "apples and pears":
            System.out.println("Stairs");
            break;
        case "lorry":
            System.out.println("truck");
            break;
        default:
            System.out.println("Don't know any more");
    }
}

```

によっては、をチェックしてのをするがあります。

えば、

```
String name;

if (A > B) {
    name = "Billy";
} else {
    name = "Jimmy";
}
```

これはのようになでできます。

```
String name = A > B ? "Billy" : "Jimmy";
```

のは、がであれば、ののにされます。がfalseの、2のがにえられます。

ブレーク

breakステートメントは、ループの for 、 while またはswitchステートメントのをします。

ループ

```
while(true) {
    if(someCondition == 5) {
        break;
    }
}
```

こののループはにされます。しかしsomeConditionがあるで5にしいとき、ループはする。

のループがカスケードされている、ものループのみがbreakをしてします。

してください...キャッチ...に

try { ... } catch (...) { ... }は、 をするためにされまます。

```
String age_input = "abc";
try {
    int age = Integer.parseInt(age_input);
    if (age >= 18) {
        System.out.println("You can vote!");
    } else {
        System.out.println("Sorry, you can't vote yet.");
    }
} catch (NumberFormatException ex) {
    System.err.println("Invalid input.  '" + age_input + "' is not a valid integer.");
}
```

これはされます

。 'abc'はなではありません。

`catch`に`finally`をすることができます。 `finally`は、がスローされたかどうかにかかわらず、にされます。

```
try { ... } catch ( ... ) { ... } finally { ... }
```

```
String age_input = "abc";
try {
    int age = Integer.parseInt(age_input);
    if (age >= 18) {
        System.out.println("You can vote!");
    } else {
        System.out.println("Sorry, you can't vote yet.");
    }
} catch (NumberFormatException ex) {
    System.err.println("Invalid input.  '" + age_input + "' is not a valid integer.");
} finally {
    System.out.println("This code will always be run, even if an exception is thrown");
}
```

これはされます

- 。 'abc'はなではありません。
- このコードは、がスローされたでもにされます

ネストされたブレイク/

ラベルステートメントをすることで、ループを`break / continue`することができます

```
outerloop:
for(...) {
    innerloop:
    for(...) {
        if(condition1)
            break outerloop;

        if(condition2)
            continue innerloop; // equivalent to: continue;
    }
}
```

Javaのラベルにはのはありません。

Javaでのステートメントの

`continue`ステートメントは、のでのステップをスキップし、のループであるためにされます。コントロールは、 `continue`ステートメントからステップインクリメントまたはデクリメントするにみます。

```
String[] programmers = {"Adrian", "Paul", "John", "Harry"};

//john is not printed out
for (String name : programmers) {
    if (name.equals("John"))
```

```
        continue;
    System.out.println(name);
}
```

`continue`は、プログラムのをきループのステップするにシフトさせることもできます。

```
Outer: // The name of the outermost loop is kept here as 'Outer'
for(int i = 0; i < 5; )
{
    for(int j = 0; j < 5; j++)
    {
        continue Outer;
    }
}
```

オンラインでなコントロールをむ <https://riptutorial.com/ja/java/topic/118/>な コントロール

153:

"varargs"メソッドのは、そのメソッドのびしがされたのをそれぞれのとしてできるようにします。メソッドでは、のに3つのASCIIピリオド ... をします。

メソッドは、それらのをのとしてけります。のはvarargsのです。はにされますただし、のを々のメソッドとしてすのではなく、びしでなをすことはできます。

の

1. Varargsはのでなければなりません。
2. メソッドには1つのVarargsしかできません。

のにうがあります。そうしないと、プログラムによってコンパイルエラーがします。

Examples

varargs パラメータの

```
void doSomething(String... strings) {  
    for (String s : strings) {  
        System.out.println(s);  
    }  
}
```

のパラメータのの3つのピリオドは、のがまたはのとしてされることをします。Varargsはののみできます。

Varargs パラメータの

varargsをメソッドのパラメータとしてすると、またはのシーケンスのいずれかをすることが出来ます。のシーケンスがされると、それらはににされます。

このは、 printVarArgArray() メソッドにされるとのシーケンスと、メソッドのコードでじようにわれるをしています。

```
public class VarArgs {  
  
    // this method will print the entire contents of the parameter passed in  
  
    void printVarArgArray(int... x) {  
        for (int i = 0; i < x.length; i++) {  
            System.out.print(x[i] + ",");  
        }  
    }  
  
    public static void main(String args[]) {  
        VarArgs obj = new VarArgs();  
    }  
}
```



```
//Using an array:
int[] testArray = new int[]{10, 20};
obj.printVarArgArray(testArray);

System.out.println(" ");

//Using a sequence of arguments
obj.printVarArgArray(5, 6, 5, 8, 6, 31);
}
}
```

```
10,20,
5,6,5,8,6,31
```

このようなメソッドをすると、コンパイルにエラーがします。

```
void method(String... a, int... b , int c){} //Compile time error (multiple varargs )

void method(int... a, String b){} //Compile time error (varargs must be the last argument
```

オンラインでをむ <https://riptutorial.com/ja/java/topic/1948/-->

154:

き

は、なOOPオブジェクトプログラミングのの1つです。は、ギリシャの "poly"と "morphs"にしています。ポリは「く」をし、モーフは「フォーム」くのをします。

をするには2つのがあります。 メソッドオーバーロードとメソッドオーバーライド。

`Interfaces`は、クラスベースのとはに、Javaでをするのです。インタフェースは、プログラムのAPIをするメソッドのリストをします。クラスは、すべてのメソッドをオーバーライドして `interface implement` するがあります。

Examples

メソッドのオーバーロード

メソッドのオーバーロードは、のオーバーロードともばれ、クラスのまたはのいずれかがなることをに、じのメソッドをつクラスのです。

コンパイラはメソッドオーバーロードのメソッドシグネチャをチェックします。

メソッドのシグネチャは3つのものからされます。

1. メソッド
2. パラメータ
3. パラメータの

これらの3つがクラスのの2つのメソッドでじ、コンパイラはしたメソッドエラーをスローします。

このタイプのは、びされるなメソッドがリストについてコンパイルにコンパイラによってされるため、またはコンパイルのとばれます。

```
class Polymorph {  
  
    public int add(int a, int b){  
        return a + b;  
    }  
  
    public int add(int a, int b, int c){  
        return a + b + c;  
    }  
  
    public float add(float a, float b){  
        return a + b;  
    }  
}
```

```

public static void main(String... args){
    Polymorph poly = new Polymorph();
    int a = 1, b = 2, c = 3;
    float d = 1.5, e = 2.5;

    System.out.println(poly.add(a, b));
    System.out.println(poly.add(a, b, c));
    System.out.println(poly.add(d, e));
}
}

```

これにより、のがられます。

```

2
6
4.000000

```

オーバーロードされたメソッドはでもでもかまいません。これはメソッドのオーバーロードにもしません。

```

public class Polymorph {

    private static void methodOverloaded()
    {
        //No argument, private static method
    }

    private int methodOverloaded(int i)
    {
        //One argument private non-static method
        return i;
    }

    static int methodOverloaded(double d)
    {
        //static Method
        return 0;
    }

    public void methodOverloaded(int i, double d)
    {
        //Public non-static Method
    }
}

```

また、メソッドのりのをした、メソッドのオーバーロードとしてできません。

```

public class Polymorph {

    void methodOverloaded(){
        //No argument and No return type
    }

    int methodOverloaded(){
        //No argument and int return type
    }
}

```

```
    return 0;
}
```

メソッドオーバーライド

メソッドのオーバーライドは、サブタイプがスーパータイプのものをオーバーライドするです。

Javaでは、これはスーパークラスでされたメソッドをオーバーライドするサブクラスにされます。Javaでは、プリミティブはすべてには `references` であり、メモリのオブジェクトへのポインタにしています。 `references` は1つのしかありません。はされたです。ただし、されたまたはそのサブタイプのいずれかのオブジェクトをすことができます。

`reference` でメソッドがびされると、しされているのオブジェクトのするメソッドがびされます。

```
class SuperType {
    public void sayHello(){
        System.out.println("Hello from SuperType");
    }

    public void sayBye(){
        System.out.println("Bye from SuperType");
    }
}

class SubType extends SuperType {
    // override the superclass method
    public void sayHello(){
        System.out.println("Hello from SubType");
    }
}

class Test {
    public static void main(String... args){
        SuperType superType = new SuperType();
        superType.sayHello(); // -> Hello from SuperType

        // make the reference point to an object of the subclass
        superType = new SubType();
        // behaviour is governed by the object, not by the reference
        superType.sayHello(); // -> Hello from SubType

        // non-overridden method is simply inherited
        superType.sayBye(); // -> Bye from SuperType
    }
}
```

えておくべきルール

サブクラスのメソッドをオーバーライドするには、オーバーライドするメソッドサブクラスのメソッドがです。

-
- プリミティブのはじりですサブクラスはクラスにしてされていますが、これはリターンとも

ばれます。

- パラメータの型と
- スーパークラスのメソッドのthrowsでされたまたはされたのサブクラスであるだけのスローすることがあります。をスローしないこともできます。パラメータはありません。例えば、void methodXint iはvoid methodXint kと同じです。
- finalメソッドやstaticメソッドをオーバーライドすることはできません。たちがうことができるのはメソッドボディだけをするだけです。

のコードにれずにクラスをすることです

```
import java.util.ArrayList;
import java.util.List;

import static java.lang.System.out;

public class PolymorphismDemo {

    public static void main(String[] args) {
        List<FlyingMachine> machines = new ArrayList<FlyingMachine>();
        machines.add(new FlyingMachine());
        machines.add(new Jet());
        machines.add(new Helicopter());
        machines.add(new Jet());

        new MakeThingsFly().letTheMachinesFly(machines);
    }
}

class MakeThingsFly {
    public void letTheMachinesFly(List<FlyingMachine> flyingMachines) {
        for (FlyingMachine flyingMachine : flyingMachines) {
            flyingMachine.fly();
        }
    }
}

class FlyingMachine {
    public void fly() {
        out.println("No implementation");
    }
}

class Jet extends FlyingMachine {
    @Override
    public void fly() {
        out.println("Start, taxi, fly");
    }

    public void bombardment() {
        out.println("Fire missile");
    }
}

class Helicopter extends FlyingMachine {
    @Override
    public void fly() {
        out.println("Start vertically, hover, fly");
    }
}
```

```
}  
}
```

a MakeThingsFlyクラスは、FlyingMachineのすべてでします。

b letTheMachinesFlyメソッドは、PropellerPlaneのようにしいクラスをするときにもなしでします。

```
public void letTheMachinesFly(List<FlyingMachine> flyingMachines) {  
    for (FlyingMachine flyingMachine : flyingMachines) {  
        flyingMachine.fly();  
    }  
}
```

それがのです。 [オープン・クローズド・プリンシプル](#)をすることができます。

メソッドではなく、FinalというキーワードがないJavaのメソッドです。デフォルトではすべてのメソッドがJavaのメソッドです。オーバーライドされるがで、じメソッドシグネチャをつ、Javaのクラスがクラスのメソッドをオーバーライドできるため、バーチャルメソッドはにおいてなをたします。

しかし、ではないいくつかのがあります。たとえば、メソッドがprivateまたはキーワードfinalでされている、メソッドはVirtualではありません。

このStackOverflowポストからメソッドをしたののをえてみましょう。 [はCとJavaではどのようにしますか](#)

```
public class A{  
    public void hello(){  
        System.out.println("Hello");  
    }  
  
    public void boo(){  
        System.out.println("Say boo");  
    }  
}  
  
public class B extends A{  
    public void hello(){  
        System.out.println("No");  
    }  
  
    public void boo(){  
        System.out.println("Say haha");  
    }  
}
```

クラスBをびしてhelloとbooをびすと、BがAとじメソッドをオーバーライドするため、として「いいえ」と「いいえ」がられます。のは、メソッドのオーバーライドをうには、クラスAのメソッドがデフォルトですべてVirtualであることをすることがです。

さらに、**abstract**キーワードをしてVirtualメソッドをできます。キーワード "abstract"でされたメソッドにはメソッドがありません。つまり、メソッドのはまだされていません。のをもうえてみましょう。ただし、**boo**メソッドが**abstract**とされているがなります。

```
public class A{
    public void hello(){
        System.out.println("Hello");
    }

    abstract void boo();
}

public class B extends A{
    public void hello(){
        System.out.println("No");
    }

    public void boo(){
        System.out.println("Say haha");
    }
}
```

Bからbooをびすと、Bはメソッドbooをし、booが "Say haha"をするので、は "Say haha"になります。

されたソースおよびさらなるみ

[CとJavaではどのようにしますか](#)

にするさらにしいをするこのらしいえをしてください

[Javaでメソッドをできますか](#)

となるタイプのオーバーライド

Java [チュートリアル](#)から

ののは、またはがくのなるまたはをすることができるにおけるをす。このは、オブジェクトプログラミングやJavaのようなにもできます。 クラスのサブクラスはのをできますが、クラスとじをいくつかできます。

このをて、なるタイプのきをしてください。

1. クラスはをせず、サブクラスはなメソッドをオーバーライドするがあります -
2. クラスは、デフォルトのをし、サブクラスはをすることができます
3. `super.methodName()`、のとして `super.methodName()` をびすことによって、クラスのにをします
4. クラスはアルゴリズムのTemplateメソッドをし、サブクラスはアルゴリズムのをきする

コードスニペット

```

import java.util.HashMap;

abstract class Game implements Runnable{

    protected boolean runGame = true;
    protected Player player1 = null;
    protected Player player2 = null;
    protected Player currentPlayer = null;

    public Game(){
        player1 = new Player("Player 1");
        player2 = new Player("Player 2");
        currentPlayer = player1;
        initializeGame();
    }

    /* Type 1: Let subclass define own implementation. Base class defines abstract method to
force
    sub-classes to define implementation
    */

    protected abstract void initializeGame();

    /* Type 2: Sub-class can change the behaviour. If not, base class behaviour is applicable
    */
    protected void logTimeBetweenMoves(Player player){
        System.out.println("Base class: Move Duration: player.PlayerActTime -
player.MoveShownTime");
    }

    /* Type 3: Base class provides implementation. Sub-class can enhance base class
implementation by calling
        super.methodName() in first line of the child class method and specific implementation
later */
    protected void logGameStatistics(){
        System.out.println("Base class: logGameStatistics:");
    }

    /* Type 4: Template method: Structure of base class can't be changed but sub-class can
some part of behaviour */
    protected void runGame() throws Exception{
        System.out.println("Base class: Defining the flow for Game:");
        while (runGame) {
            /*
            1. Set current player
            2. Get Player Move
            */
            validatePlayerMove(currentPlayer);
            logTimeBetweenMoves(currentPlayer);
            Thread.sleep(500);
            setNextPlayer();
        }
        logGameStatistics();
    }
    /* sub-part of the template method, which define child class behaviour */
    protected abstract void validatePlayerMove(Player p);

    protected void setRunGame(boolean status){
        this.runGame = status;
    }

    public void setCurrentPlayer(Player p){
        this.currentPlayer = p;
    }

```



```

    }
    public void setNextPlayer(){
        if (currentPlayer == player1) {
            currentPlayer = player2;
        }else{
            currentPlayer = player1;
        }
    }
    public void run(){
        try{
            runGame();
        }catch(Exception err){
            err.printStackTrace();
        }
    }
}

class Player{
    String name;
    Player(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
}

/* Concrete Game implementation */
class Chess extends Game{
    public Chess(){
        super();
    }
    public void initializeGame(){
        System.out.println("Child class: Initialized Chess game");
    }
    protected void validatePlayerMove(Player p){
        System.out.println("Child class: Validate Chess move:" + p.getName());
    }
    protected void logGameStatistics(){
        super.logGameStatistics();
        System.out.println("Child class: Add Chess specific logGameStatistics:");
    }
}

class TicTacToe extends Game{
    public TicTacToe(){
        super();
    }
    public void initializeGame(){
        System.out.println("Child class: Initialized TicTacToe game");
    }
    protected void validatePlayerMove(Player p){
        System.out.println("Child class: Validate TicTacToe move:" + p.getName());
    }
}

public class Polymorphism{
    public static void main(String args[]){
        try{

            Game game = new Chess();
            Thread t1 = new Thread(game);

```

```

        t1.start();
        Thread.sleep(1000);
        game.setRunGame(false);
        Thread.sleep(1000);

        game = new TicTacToe();
        Thread t2 = new Thread(game);
        t2.start();
        Thread.sleep(1000);
        game.setRunGame(false);

    }catch(Exception err){
        err.printStackTrace();
    }
}
}

```

```

Child class: Initialized Chess game
Base class: Defining the flow for Game:
Child class: Validate Chess move:Player 1
Base class: Move Duration: player.PlayerActTime - player.MoveShownTime
Child class: Validate Chess move:Player 2
Base class: Move Duration: player.PlayerActTime - player.MoveShownTime
Base class: logGameStatistics:
Child class: Add Chess specific logGameStatistics:

Child class: Initialized TicTacToe game
Base class: Defining the flow for Game:
Child class: Validate TicTacToe move:Player 1
Base class: Move Duration: player.PlayerActTime - player.MoveShownTime
Child class: Validate TicTacToe move:Player 2
Base class: Move Duration: player.PlayerActTime - player.MoveShownTime
Base class: logGameStatistics:

```

オンラインでをむ <https://riptutorial.com/ja/java/topic/980/>

155: なオブジェクト

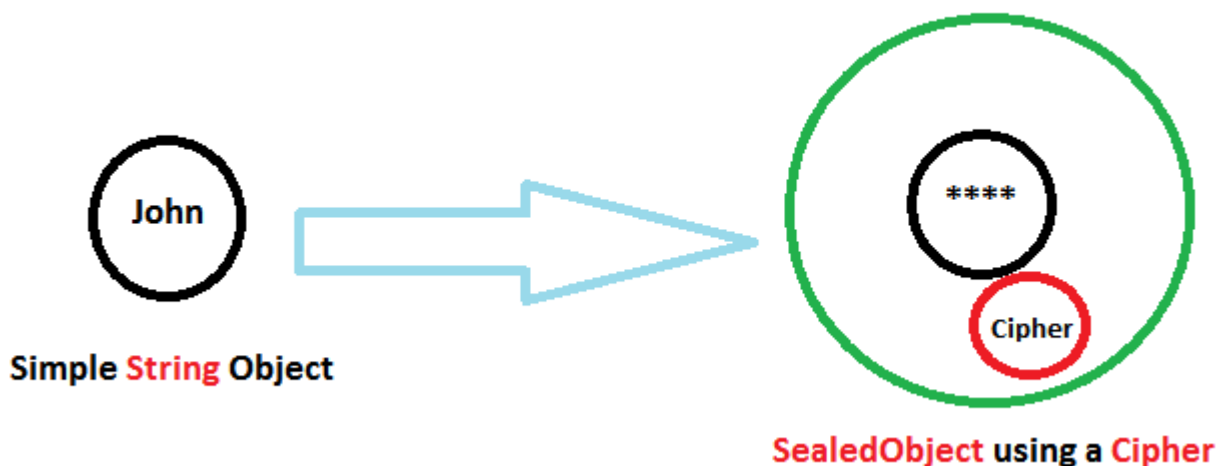
- SealedObject sealedObject = しい SealedObject obj、 cipher;
- SignedObject signedObject = しい SignedObject obj、 signingKey、 signingEngine;

Examples

SealedObject javax.crypto.SealedObject

このクラスをすると、プログラマーはオブジェクトをし、アルゴリズムをしてそのをすることができます。

シリアルライズなオブジェクトがあれば、AES、DESなどのアルゴリズムをしてシリアルされたコンテンツをシリアルライズされたフォーマットつまり「ディープコピー」でカプセルし、シリアルされたコンテンツをシールする**SealedObject**をできますその。されたコンテンツは、なをしてするアルゴリズムをしてし、シリアルしてのオブジェクトをすることができます。

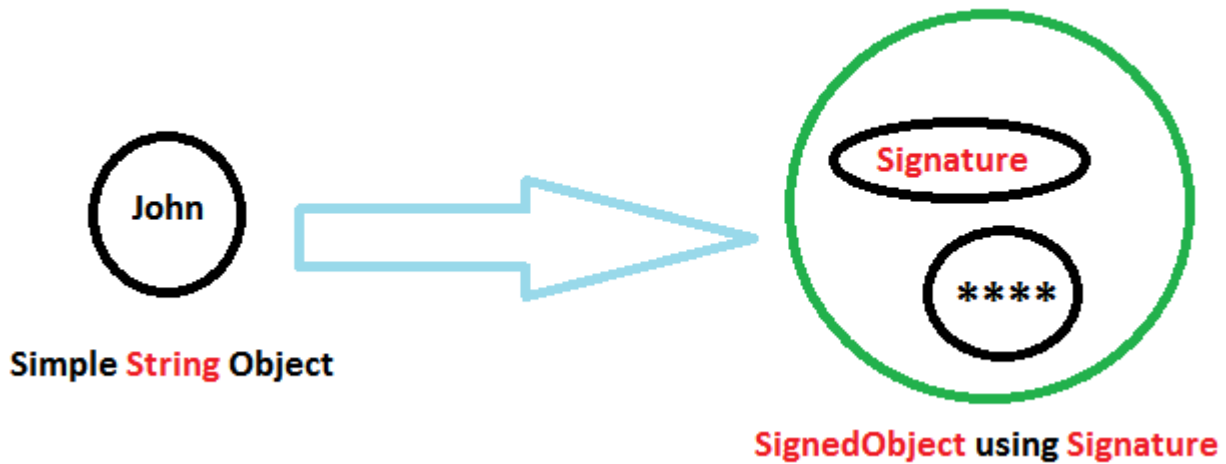


```
Serializable obj = new String("John");
// Generate key
KeyGenerator kgen = KeyGenerator.getInstance("AES");
kgen.init(128);
SecretKey aesKey = kgen.generateKey();
Cipher cipher = Cipher.getInstance("AES");
cipher.init(Cipher.ENCRYPT_MODE, aesKey);
SealedObject sealedObject = new SealedObject(obj, cipher);
System.out.println("sealedObject-" + sealedObject);
System.out.println("sealedObject Data-" + sealedObject.getObject(aesKey));
```

SignedObject java.security.SignedObject

SignedObjectは、がされずになわれないことのない、のランタイムオブジェクトをするためのクラスです。

よりには、SignedObjectには、のシリアルズオブジェクト、されるきオブジェクトとそのがまれます。



```
//Create a key
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA", "SUN");
SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
keyGen.initialize(1024, random);
// create a private key
PrivateKey signingKey = keyGen.generateKeyPair().getPrivate();
// create a Signature
Signature signingEngine = Signature.getInstance("DSA");
signingEngine.initSign(signingKey);
// create a simple object
Serializable obj = new String("John");
// sign our object
SignedObject signedObject = new SignedObject(obj, signingKey, signingEngine);

System.out.println("signedObject-" + signedObject);
System.out.println("signedObject Data-" + signedObject.getObject());
```

オンラインでなオブジェクトをむ <https://riptutorial.com/ja/java/topic/5528/なオブジェクト>

156: な

き

`CompletableFuture`は、Java SE 8にされたクラスであり、Java SE 5の`Future`インタフェースをしています。`Future`インタフェースをサポートするだけでなく、のにコールバックをにするくのメソッドがされています。

Examples

ブロックメソッドをにする

のでは、Webページをしてテキストのさをえるために、にじて22かかります。どんなスレッドがびされても、それはそのブロックされます。また、でながされます。

```
public static long blockingGetWebPageLength(String urlString) {
    try (BufferedReader br = new BufferedReader(new InputStreamReader(new
        URL(urlString).openConnection().getInputStream()))) {
        StringBuilder sb = new StringBuilder();
        String line;
        while ((line = br.readLine()) != null) {
            sb.append(line);
        }
        return sb.toString().length();
    } catch (IOException ex) {
        throw new RuntimeException(ex);
    }
}
```

これは、ブロッキングメソッドびしをのスレッドにすることによってすぐになるメソッドにします。デフォルトでは、`supplyAsync`メソッドはプールでサプライヤをします。はオプションのサービスパラメータをしたは、プールのスレッドをいたしているかもしれないので、これはブロッキングメソッドのために、おそらくいではありません。

```
static private ExecutorService service = Executors.newCachedThreadPool();

static public CompletableFuture<Long> asyncGetWebPageLength(String url) {
    return CompletableFuture.supplyAsync(() -> blockingGetWebPageLength(url), service);
}
```

ファンクションでをするには、`thenAccept`などのに、サプライヤのでびされるlamdaをけるメソッドをするがあります。また、したのあるをするために、にするか、またはメソッドをすることもです。

```
public static void main(String[] args) {

    asyncGetWebPageLength("https://stackoverflow.com/")
}
```

```

        .thenAccept(1 -> {
            System.out.println("Stack Overflow returned " + 1);
        })
        .exceptionally((Throwable throwable) -> {
            Logger.getLogger("myclass").log(Level.SEVERE, "", throwable);
            return null;
        });
    }
}

```

CompletableFutureのな

のでは、`calculateShippingPrice`メソッドは、をすのサーバにすることである。

`CompletableFuture`をしてこれをにモデリングすることで、このメソッドでなるをけることができますつまり、パッケージの。

```

public static void main(String[] args) {
    int price = 15; // Let's keep it simple and work with whole number prices here
    int weightInGrams = 900;

    calculateShippingPrice(weightInGrams) // Here, we get the future
        .thenAccept(shippingPrice -> { // And then immediately work on it!
            // This fluent style is very useful for keeping it concise
            System.out.println("Your total price is: " + (price + shippingPrice));
        });
    System.out.println("Please stand by. We are calculating your total price.");
}

public static CompletableFuture<Integer> calculateShippingPrice(int weightInGrams) {
    return CompletableFuture.supplyAsync(() -> {
        // supplyAsync is a factory method that turns a given
        // Supplier<U> into a CompletableFuture<U>

        // Let's just say each 200 grams is a new dollar on your shipping costs
        int shippingCosts = weightInGrams / 200;

        try {
            Thread.sleep(2000L); // Now let's simulate some waiting time...
        } catch (InterruptedException e) { /* We can safely ignore that */ }

        return shippingCosts; // And send the costs back!
    });
}

```

オンラインでなをむ <https://riptutorial.com/ja/java/topic/10935/な>

157: コマンド

Examples

シャットダウンフックの

には、いたシステムリソースをするなど、プログラムがしたときにするコードがになることがあります。 `addShutdownHook` メソッドでプログラムがしたときにスレッドをさせることができます

```
Runtime.getRuntime().addShutdownHook(new Thread(() -> {  
    ImportantStuff.someImportantIOStream.close();  
}));
```

オンラインでコマンドをむ <https://riptutorial.com/ja/java/topic/7304/コマンド>

158:

き

Javaのは、をうためのなです。

でできるのについては、を。

Examples

オペレータの

あるにのがまれている、なるでみるがあります。例えば、 $1 + 2 \times 3$ は2つのでみることができます。

- 1. 1 と 2 をえ、に 3 をけます。これはえをえる 9 。かっこをすると、 $(1 + 2) \times 3$ ます。
- 2. 2 と 3 をしたに 1 をえます。これはえをえる 7 。かっこをすると、 $1 + (2 \times 3)$ ます。

では、は2のでもむことがコンベンションです。なは、とがとのにわれることです。よりんだがされるとき、は「」されたににとってか、またはさをするためにがされます。どちらのでも、をえるのは、のとにします。

Javaには、されているのについて、をみるにするじながあります。

に、はにされる。のをしてください。

例えば

```
1 + 2 * 3
```

$+$ のは $*$ のよりもいため、のは 9 ではなく 7 です。

	プライマリ		
インスタンスの フィールドアクセス アクセス メソッドびし メソッド	. (expr) new . プライマリ [expr] プライマリ (expr、 ...) primary :: name	15	からへ
ポストインクリメント	expr ++、expr --	14	-

	Iプライマリ		
プリインクリメント キャスト ¹	++ expr、 -- expr、 + exprは、 -expr、 ~expr、 ! expr、 () expr	13	- からへ からへ
	* /	12	からへ
	+ -	11	からへ
シフト	<< >> >>>	10	からへ
リレーショナル	< > <= >= instanceof	9	からへ
	===	8	からへ
ビットAND	そして、	7	からへ
ビットごとの	^	6	からへ
ビットOR		5	からへ
AND	&&	4	からへ
OR		3	からへ
1		2	からへ
りて ラムダ ¹	=====<<=>=>>==^== ->	1	からへ

¹ラムダのは、キャスト、またはき3の3のとしてもするため、です。

は、プリミティブまたはStringをするであり、そのはコンパイルにリテラルにされます。はをスローせずにするがあり、のものだけでされているがあります。

- プリミティブとリテラル。
- はプリミティブまたはStringキャストされます。
- の +、 -、 ~ !。
- *、 /、 %、 +、 -、 << >>、 >>>、 <、 <=、 >、 >=、 ==、 !=、 &、 ^、 |、 && と ||。
- ? :。
- の。
- をするな。 は、 final とされたで、のはです。

- `<TypeName> . <Identifier>`の`<TypeName> . <Identifier>`はをします。

のリストには、`++`と`--`、`class`と`instanceof`、メソッドびし、およびなやフィールドのはされています。

`String`のは`"interned"` `String`になり、のはFPセマンティクスでされます。

の

は、のをできるであればどこでもできます。しかしながら、それらはのにおいてなをする。

`switch`の`case`ラベルにはがです。え

```
switch (someValue) {
    case 1 + 1:           // OK
    case Math.min(2, 3): // Error - not a constant expression
        doSomething();
}
```

ののがである、はプリミティブなナローをできます。これは、のがなのであればされます。 [JLS 5.1.3](#)と[5.2](#)を

```
byte b1 = 1 + 1;           // OK - primitive narrowing conversion.
byte b2 = 127 + 1;         // Error - out of range
byte b3 = b1 + 1;          // Error - not a constant expression
byte b4 = (byte) (b1 + 1); // OK
```

`do`、`while`または`for`としてをすると、にします。え

```
while (false) {
    doSomething();           // Error - statement not reachable
}
boolean flag = false;
while (flag) {
    doSomething();          // OK
}
```

これはされないことにしてください `if`、Javaコンパイラができますステートメント。 `then`または `else`のブロック `if`であることをを。これは、CおよびC++でのきコンパイルのJavaのアナログです。

に、イニシャライザをつクラスまたはインターフェイスの `static final` フィールドがくされます。したがって、クラスグラフにサイクルがあっても、されたでこれらののがされることがされます。

は、 [JLS 15.28](#)をしてください。。

の

Javaはのにつてされます。

- オペランドはからにされます。
- オペレータのオペランドは、オペレータのでされます。
- は、のによってされます。
- リストはからにされます。

な

のでは、

```
int i = method1() + method2();
```

のはのとおりです。

1. `=`のオペランドは、`i`のアドレスにされます。
2. `+`ののオペランド `method1()` がされます。
3. `+ method2()` のオペランドがされます。
4. `+`がされます。
5. `=`がされ、のが`i`にされます。

びしのがな、`method1`のびしは`method2`のびしのにすることが`method2`ます。

をつをした

のでは、

```
int i = 1;
intArray[i] = ++i + 1;
```

のはのとおりです。

1. `=`のオペランドがされます。これにより、`intArray[1]` アドレスがられます。
2. プリインクリメントがされます。これは`i`に`1`をえ、`2`とします。
3. `+`ののオペランドがされます。
4. `+`は、`2 + 1 -> 3`とされます。
5. `=`がされ、`intArray[1]` `3`がされます。

オペランドのでことに`=`にされ、それはののをけない`++i`。

- [JLS 15.7 - オーダー](#)

の

Javaのは、をうためのなです。ここではいくつかのをします。

```
1 // A simple literal is an expression
```

```

1 + 2           // A simple expression that adds two numbers
(i + j) / k     // An expression with multiple operations
(flag) ? c : d  // An expression using the "conditional" operator
(String) s      // A type-cast is an expression
obj.test()      // A method call is an expression
new Object()    // Creation of an object is an expression
new int[]       // Creation of an object is an expression

```

に、はのでされます。

- のはのものでされます。
 - な。 `someIdentifier`
 - された。 `MyClass.someField`
- からなる
 - リテラル; えば `1`、`1.0`、`'X'` `"hello"`、`false` と `null`
 - クラスのリテラル。 `MyClass.class`
 - `this` と `<TypeName> . this`
 - の。 えば `( a + b )`
 - クラスインスタンス。 `new MyClass(1, 2, 3)`
 - インスタンス。 `new int[3]`
 - フィールドアクセス。 `obj.someField` または `this.someField`
 - アクセス。 えば `vector[21]`
 - メソッドびし。 えば、 `obj.doIt(1, 2, 3)`
 - メソッド **Java 8**。 `MyClass::doIt`
- 。 えば `!a` または `i++`
- バイナリ。 えば `a + b` または `obj == null`
- の。 えば `(obj == null) ? 1 : obj.getCount()`
- ラムダ **Java 8**。 `obj -> obj.getCount()`

さまざまなのは、のトピックにあります。

- のトピックでは、`、`、`、` のについてします。
- ラムダのトピックでは、ラムダとメソッドについてします。
- クラスとオブジェクトのトピックでは、クラスインスタンスのについてします。
- トピックでは、アクセスとインスタンスについてします。
- リテラルトピックでは、リテラルのさまざまなについてします。

のタイプ

ほとんどの、は、コンパイルにしてそのをすることができるをちます。これらは、 とされます。

しかし、Java 8ではのはポリであるがあります。

- の
- クラスインスタンス
- メソッドびし

- メソッド
-
- ラムダ

がポリである、そののはターゲットのをけるがあります。それがのためにわれているのか。

の

のはそのとりてです。はヒープがしたです。たとえば、「でない」のがにされたりされたりしているためです。

ステートメント

のくのとはなり、Javaではにをとしてすることはできません。えは

```
public void compute(int i, int j) {
    i + j;    // ERROR
}
```

このようなをしたはできないため、のでプログラムのにをほすことができないため、Javaのはそのようながいかっているというをつた。

ただし、これはすべてののにてはまるわけではありません。のサブセットは、にはステートメントとしてです。このセットは、

- オペレーション・アンド・アサインド・アサインメントをむ。
- とのインクリメントとデクリメントの。
- メソッドびし `void` または `non-void`。
- クラスインスタンス。

オンラインでをむ <https://riptutorial.com/ja/java/topic/8167/>

159: いハッシュマップ

き

いハッシュマップの

Examples

いハッシュマップの

キーポイント。

- の。
- そのキーにするいのみをします。

いによってされるオブジェクトがガーベッジにされます。 GCはそのにメモリがになるまでしません。

ハッシュマップとハッシュマップのい

Javaメモリマネージャがキーとしてされたオブジェクトへのいをもはやっていない、マップのエントリはWeakHashMapでされます。

-

```
public class WeakHashMapTest {
    public static void main(String[] args) {
        Map hashMap= new HashMap();

        Map weakHashMap = new WeakHashMap();

        String keyHashMap = new String("keyHashMap");
        String keyWeakHashMap = new String("keyWeakHashMap");

        hashMap.put(keyHashMap, "Ankita");
        weakHashMap.put(keyWeakHashMap, "Atul");
        System.gc();
        System.out.println("Before: hash map value:"+hashMap.get("keyHashMap")+" and weak hash
map value:"+weakHashMap.get("keyWeakHashMap"));

        keyHashMap = null;
        keyWeakHashMap = null;

        System.gc();

        System.out.println("After: hash map value:"+hashMap.get("keyHashMap")+" and weak hash
map value:"+weakHashMap.get("keyWeakHashMap"));
    }
}
```

サイズのい**HashMapWeakHashMap**

HashMapオブジェクトのsizeメソッドを呼び出すと、そのキーと値のペアが返されます。sizeは、removeメソッドがHashMapオブジェクトに対して呼び出された後にのみ呼び出されます。

WeakHashMapはいつでもキーを削除することがあるため、WeakHashMapはスレッドが安全なエントリを持っているかのように見えます。したがって、sizeメソッドはそれと一致しない場合があります。WeakHashMapでは、サイズが正確ではありません。

オンラインでハッシュマップを学ぶ <https://riptutorial.com/ja/java/topic/10749/ハッシュマップ>

160: ちとち

Examples

PriorityQueueの

PriorityQueueはデータです。SortedSetとに、PriorityQueueはにづいてをソートします。のいがされます。PriorityQueueのは、データののをするメソッドをcomparable、comparator comparableまたはcomparatorインタフェースをcomparableがあります。

```
//The type of the PriorityQueue is Integer.
PriorityQueue<Integer> queue = new PriorityQueue<Integer>();

//The elements are added to the PriorityQueue
queue.addAll( Arrays.asList( 9, 2, 3, 1, 3, 8 ) );

//The PriorityQueue sorts the elements by using compareTo method of the Integer Class
//The head of this queue is the least element with respect to the specified ordering
System.out.println( queue ); //The Output: [1, 2, 3, 9, 3, 8]
queue.remove();
System.out.println( queue ); //The Output: [2, 3, 3, 9, 8]
queue.remove();
System.out.println( queue ); //The Output: [3, 8, 3, 9]
queue.remove();
System.out.println( queue ); //The Output: [3, 8, 9]
queue.remove();
System.out.println( queue ); //The Output: [8, 9]
queue.remove();
System.out.println( queue ); //The Output: [9]
queue.remove();
System.out.println( queue ); //The Output: []
```

FIFOキューとしてのLinkedList

java.util.LinkedListクラスでは、しながら、java.util.Listのなであるjava.util.QueueあまりにするインターフェイスFIFOれしの。

のでは、offer()メソッドをして、をLinkedListします。このはenqueueとされます。のwhileループでは、はFIFOにづいてQueueからされます。このはdequeueとされます。

```
Queue<String> queue = new LinkedList<String>();

queue.offer( "first element" );
queue.offer( "second element" );
queue.offer( "third element" );
queue.offer( "fourth. element" );
queue.offer( "fifth. element" );

while ( !queue.isEmpty() ) {
    System.out.println( queue.poll() );
}
```


このコードのはのとおりで。

```
first element
second element
third element
fourth element
fifth element
```

にられるように、1のされた「1の」がにされ、「2の」が2のなどでされる。

スタック

スタックとはですか

Javaでは、StacksはオブジェクトのLIFO Last In、First Outデータです。

Stack API

JavaにはのメソッドをむStack APIがまれています

Stack()	//Creates an empty Stack	
isEmpty()	//Is the Stack Empty?	Return Type: Boolean
push(Item item)	//push an item onto the stack	
pop()	//removes item from top of stack	Return Type: Item
size()	//returns # of items in stack	Return Type: Int

```
import java.util.*;

public class StackExample {

    public static void main(String args[]) {
        Stack st = new Stack();
        System.out.println("stack: " + st);

        st.push(10);
        System.out.println("10 was pushed to the stack");
        System.out.println("stack: " + st);

        st.push(15);
        System.out.println("15 was pushed to the stack");
        System.out.println("stack: " + st);

        st.push(80);
        System.out.println("80 was pushed to the stack");
        System.out.println("stack: " + st);

        st.pop();
        System.out.println("80 was popped from the stack");
        System.out.println("stack: " + st);
    }
}
```

```

    st.pop();
    System.out.println("15 was popped from the stack");
    System.out.println("stack: " + st);

    st.pop();
    System.out.println("10 was popped from the stack");
    System.out.println("stack: " + st);

    if(st.isEmpty())
    {
        System.out.println("empty stack");
    }
}
}

```

り

```

stack: []
10 was pushed to the stack
stack: [10]
15 was pushed to the stack
stack: [10, 15]
80 was pushed to the stack
stack: [10, 15, 80]
80 was popped from the stack
stack: [10, 15]
15 was popped from the stack
stack: [10]
10 was popped from the stack
stack: []
empty stack

```

BlockingQueue

BlockingQueueはインターフェイスで、キューからデキューしようとするときキューをにしたり、キューにをエンキューしようとしたとき、キューがすでにいっぱいになったりするとブロックされます。のキューからデキューしようとするスレッドは、のスレッドがアイテムをキューにするまでブロックされます。1つまたはのアイテムをデキューするか、キューをにクリアすることによって、のスレッドがキューにスペースをるまで、フル・キューのアイテムをエンキューしようとしているスレッドはブロックされます。

BlockingQueueメソッドは、すぐにはできないをするさまざまで、4つのでされますが、のあるではできるかもしれません1つはをスローし、2つはなをしますがするまでのスレッドをにブロックし、4のブロックは、あきらめるにののみをブロックします。

	をスローする		ブロック	タイムアウト
インサート	add	オファ—e	pute	オファ—e、
する	remove	poll	る	、
べる		ピーク	N / A	N / A

BlockingQueue またはすることができます。BlockingQueueは、でされるものです。

```
BlockingQueue<String> bQueue = new ArrayBlockingQueue<String>(2);
```

キューのサイズがされているといい、putメソッドのびしはすべてブロックされます。

のキューは、なしでされるキューで、にはデフォルトでInteger.MAX_VALUEでされます。

BlockingQueue なはのとおりで。

1. ArrayBlockingQueue
2. LinkedBlockingQueue
3. PriorityBlockingQueue

に、 ArrayBlockingQueue をてみましょう。

```
BlockingQueue<String> bQueue = new ArrayBlockingQueue<>(2);
bQueue.put("This is entry 1");
System.out.println("Entry one done");
bQueue.put("This is entry 2");
System.out.println("Entry two done");
bQueue.put("This is entry 3");
System.out.println("Entry three done");
```

これはされます

```
Entry one done
Entry two done
```

そして、スレッドは2ののにブロックされます。

キューインターフェイス

Queueは、にをするためのコレクションです。キューは、をFIFOれしでけしますが、ではありません。

キューのは、またはポーリングのびしによってされるです。FIFOキューでは、すべてのしいがキューのにされます。

キューインタフェース

```
public interface Queue<E> extends Collection<E> {
    boolean add(E e);

    boolean offer(E e);

    E remove();

    E poll();
```

```

    E element();

    E peek();
}

```

Queueメソッドは2つのでします。

- がしたはがスローされます。
- otherは、がしたになをしますによってはnullまたはfalse。

のタイプ	をスローする	なをす
インサート	add(e)	offer(e)
する	remove()	poll()
べる	element()	peek()

デキ

Dequeは、キューのまたはにをできる「Dequeキュー」です。キューは、キューのにをすることしかできません。

DequeはQueueインタフェースをします。つまり、のメソッドがっていることをしますが、Dequeインタフェースは、キューにをたせるためののメソッドをします。キューがどのようにするかをっていれば、のはにはのためにします。これらのメソッドはをめるためです。

	な
getFirst()	せずにキューのののをします。
getLast()	キューのののをせずにします。
addFirst(E e)	アイテムをキューのにします。
addLast(E e)	アイテムをキューのにします。
removeFirst()	キューのにあるのをします。
removeLast()	キューのにあるのをします。

もちろん、offer、poll、peekオプションもじですが、はなく、なでします。らがここですしているかをしてもがありません。

のとアクセス

Dequeののにをするには、`add()` メソッドをびします。また、することができます `addFirst()` と `addLast()` キューのとにをするを、。

```
Deque<String> dequeA = new LinkedList<>();

dequeA.add("element 1");           //add element at tail
dequeA.addFirst("element 2");      //add element at head
dequeA.addLast("element 3");       //add element at tail
```

をキューからりすことなく、キューのにあるをくことができます。これは、`element()` メソッドをしてわれます。また、Deque とのをす `getFirst()` メソッドと `getLast()` メソッドをすることもできます。それはどのようにえるかです

```
String firstElement0 = dequeA.element();
String firstElement1 = dequeA.getFirst();
String lastElement = dequeA.getLast();
```

の

キューからをするには、`remove()`、`removeFirst()` および `removeLast()` メソッドを `removeLast()` ます。いくつかのがあります

```
String firstElement = dequeA.remove();
String firstElement = dequeA.removeFirst();
String lastElement = dequeA.removeLast();
```

オンラインでちとちをむ <https://riptutorial.com/ja/java/topic/7196/ちとち>

161: でまる

き

Javaでは、のは100A、25Kなどのでめることはできません。その、コードに_アンダースコアまたはされたパターンをしてチェックすることができます。

Examples

のついた

```
public enum BookCode {
    _10A("Simon Haykin", "Communication System"),
    _42B("Stefan Hakins", "A Brief History of Time"),
    E1("Sedra Smith", "Electronics Circuits");

    private String author;
    private String title;

    BookCode(String author, String title) {
        this.author = author;
        this.title = title;
    }

    public String getName() {
        String name = name();
        if (name.charAt(0) == '_') {
            name = name.substring(1, name.length());
        }
        return name;
    }

    public static BookCode of(String code) {
        if (Character.isDigit(code.charAt(0))) {
            code = "_" + code;
        }
        return BookCode.valueOf(code);
    }
}
```

オンラインででまるをむ <https://riptutorial.com/ja/java/topic/10719/でまる>

162: コード

Examples

UTF-8でエンコードされたファイルからテキストをみる

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;

public class ReadingUTF8TextFile {

    public static void main(String[] args) throws IOException {
        //StandardCharsets is available since Java 1.7
        //for ealier version use Charset.forName("UTF-8");
        try (BufferedWriter wr = Files.newBufferedWriter(Paths.get("test.txt"),
StandardCharsets.UTF_8)) {
            wr.write("Strange cyrillic symbol Ъ");
        }
        /* First Way. For big files */
        try (BufferedReader reader = Files.newBufferedReader(Paths.get("test.txt"),
StandardCharsets.UTF_8)) {

            String line;
            while ((line = reader.readLine()) != null) {
                System.out.print(line);
            }

            System.out.println(); //just separating output

            /* Second way. For small files */
            String s = new String(Files.readAllBytes(Paths.get("test.txt")),
StandardCharsets.UTF_8);
            System.out.print(s);
        }
    }
}
```

UTF-8でファイルにテキストをきむ

```
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;

public class WritingUTF8TextFile {

    public static void main(String[] args) throws IOException {
        //StandardCharsets is available since Java 1.7
        //for ealier version use Charset.forName("UTF-8");
        try (BufferedWriter wr = Files.newBufferedWriter(Paths.get("test2.txt"),
```

```
StandardCharsets.UTF_8)) {  
    wr.write("Cyrillic symbol Ъ");  
}  
}  
}
```

UTF-8でのバイトの

```
import java.nio.charset.StandardCharsets;  
import java.util.Arrays;  
  
public class GetUtf8BytesFromString {  
  
    public static void main(String[] args) {  
        String str = "Cyrillic symbol Ъ";  
        //StandardCharsets is available since Java 1.7  
        //for ealier version use Charset.forName("UTF-8");  
        byte[] textInUtf8 = str.getBytes(StandardCharsets.UTF_8);  
  
        System.out.println(Arrays.toString(textInUtf8));  
    }  
}
```

オンラインでコードをむ <https://riptutorial.com/ja/java/topic/2735/コード>

き

`java.lang.String` は、プログラムにされているテキストです。はJavaのプリミティブなデータではありませんが、Javaプログラムではにです。

Javaでは、はできません。つまり、できません。クリックして[ここでは](#)のよりなのために。

Javaはなので、`String`をするすべてのメソッドはしい`String`オブジェクトをします。の`String`はされません。これには、CおよびC++プログラマがターゲットの`String`オブジェクトをするとされるおよびがまれます。

コンパイルにができない2つの`String`オブジェクトをするは、`String`わりに[StringBuilder](#)をします。このテクニックは、`StringBuilder`がであるため、しい`String`オブジェクトをしてするよりもれています。

[StringBuffer](#)は、`String`オブジェクトをするためにもできます。ただし、このクラスはスレッドセーフであるようにされており、のにミューテックスをするため、パフォーマンスがします。をするときはスレッドセーフであるはほとんどないので、`StringBuilder`をすることをおめします。

をのとしてできるは、+をするがよいでしょう。Javaコンパイラは、+をむを`String.concat(...)`または[StringBuilder](#)をしてなのにしします。`StringBuilder`にするためのアドバイスは、にのがまれるにのみされます。

をにしないでください。かがのアプリケーションのメモリダンプをできる、の`String`オブジェクトをすべてつけてそのをみることができます。これには、でガベージコレクションを行っている`String`オブジェクトがまれます。これがであれば、いのデータをしたらすぐにするがあります。これはなので、`String`オブジェクトではできません。したがって、いのデータをするには`char[]`オブジェクトをし、したらそれらをしますたとえば、`'\000'`でできるなど。

すべての`String`インスタンスは、リテラルにするインスタンスでも、ヒープにされます。リテラルにするなことは、JVMはしいつまり、じでされるすべてのリテラルがの`String`オブジェクトでされることをすることですこのはJLSでされています。これは、JVMクラスローダーによってされます。クラスローダがクラスをロードすると、クラスでされるリテラルがスキャンされ、プールにこのリテラルのレコードがすでにするかどうかがされますリテラルをキーとしてします。リテラルのエントリがすでにするは、そのリテラルのペアとしてされている`String`インスタンスへのがされます。それのは、しい`String`インスタンスがされ、リテラルキーとしてされるのインスタンスへのがプールにされます。 [のインターン](#)もしてください。

プールはJavaヒープにされ、のガベージコレクションのです。

Java SE 7

Java 7よりのJavaのリリースでは、プールは "PermGen" とばれるヒープのなににされていました。このはたまにしかされませんでした。

Java SE 7

Java 7では、プールが「PermGen」からされました。

リテラルは、それらをするメソッドからにであることにしてください。これは、コードがガベージコレクションされているにのみ、する `String` オブジェクトがガベージコレクションされることをします。

Java 8までは、`String` オブジェクトはUTF-16 charcharあたり2バイトとしてされています。Java 9には、がバイトLATIN-1またはUTF-16としてエンコードされているかどうかをすエンコードフラグフィールドをつバイトとして `String` をするがあります。

Examples

の

をするには、`String` オブジェクトの `equals` メソッドまたは `equalsIgnoreCase` メソッドをするがあります。

たとえば、のスニペットは、`String` の2つのインスタンスがすべてのでしいかどうかをします。

```
String firstString = "Test123";
String secondString = "Test" + 123;

if (firstString.equals(secondString)) {
    // Both Strings have the same content.
}
```

ライブデモ

このでは、とをせずにします。

```
String firstString = "Test123";
String secondString = "TEST123";

if (firstString.equalsIgnoreCase(secondString)) {
    // Both Strings are equal, ignoring the case of the individual characters.
}
```

ライブデモ

ことにしてください `equalsIgnoreCase`、できません `Locale`。たとえば、で "Taki" と "TAKI" という2つのをすると、それらはしいです。しかし、トルコでは、らはなっていますトルコでは、の `I` は `ı`。このような、のを `Locale` またはにし、`equals` とすること `equals` です。

```
String firstString = "Taki";
String secondString = "TAKI";

System.out.println(firstString.equalsIgnoreCase(secondString)); //prints true

Locale locale = Locale.forLanguageTag("tr-TR");

System.out.println(firstString.toLowerCase(locale).equals(
    secondString.toLowerCase(locale))); //prints false
```

[ライブデモ](#)

==をしてをしないでください

すべてのがインターンされていることをできないり、==または!=をしてをするべきではありません。これらのはにはをテストし、のStringオブジェクトがじStringをすことができるので、これはったえをえるがあります。

そのわりに、String.equals(Object)メソッドをします。このメソッドは、Stringオブジェクトをそのについてします。については、[Pitfall==をしてをする](#)をしてください。

switchのの

Java SE 7

Java 1.7、をswitchのリテラルとすることはです。Stringがnullでないことをします。そうでない、にNullPointerExceptionがスローされます。はString.equals とをしますをしてされます。

```
String stringToSwitch = "A";

switch (stringToSwitch) {
    case "a":
        System.out.println("a");
        break;
    case "A":
        System.out.println("A"); //the code goes here
        break;
    case "B":
        System.out.println("B");
        break;
    default:
        break;
}
```

[ライブデモ](#)

との

`String` をとするは、`equals` のにをして、の `String` が `null` に `NullPointerException` ないようにすること
ができ `null` 。

```
"baz".equals(foo)
```

`foo` が `null`、`foo.equals("baz")` は `NullPointerException` をスローし `null`、`"baz".equals(foo)` は `false` と
され `false` 。

Java SE 7

よりみやすいのは、`Objects.equals()` をすること `Objects.equals()`。これは、`Objects.equals(foo, "baz")` のパラメータにしてヌルチェックをします。

に `NullPointerExceptions` をするがいかどうか、またはをするがいかどうかはのがあります [ここ](#)と
[ここ](#)をしてください。

の

`String` クラスは、`String.compareTo` メソッドで `Comparable<String>` をしていますこののでした
`String.compareTo` です。これにより、`String` オブジェクトのけがをしたになります。`String` クラス
は、をしないうソートにした `CASE_INSENSITIVE_ORDER` という `Comparator<String>` をします。

インターナショナルストリングとの

Java [JLS 3.10.6](#) には、のようにされています。

さらに、リテラルは、に `String` クラスのじインスタンスをします。これは、リテラル
またはよりにはのであるが、メソッド `String.intern` をしてのインスタンスをするため
にインターンされるため `String.intern`。 "

つまり、`==` をして2つのリテラルへのをすることはです。さらに、`String.intern()` メソッドをし
てされた `String` オブジェクトへのについてもです。

えば

```
String strObj = new String("Hello!");
String str = "Hello!";

// The two string references point two strings that are equal
if (strObj.equals(str)) {
    System.out.println("The strings are equal");
}
```

```
// The two string references do not point to the same object
if (strObj != str) {
    System.out.println("The strings are not the same object");
}

// If we intern a string that is equal to a given literal, the result is
// a string that has the same reference as the literal.
String internedStr = strObj.intern();

if (internedStr == str) {
    System.out.println("The interned string and the literal are the same object");
}
```

では、インターナショナル・メカニズムは、としてなすべてのインターナショナル・ストリングをむハッシュ・テーブルをします。Stringでintern()をびすと、メソッドはハッシュテーブルのオブジェクトをします。

- がつかったは、そのがとしてされます。
- それのは、のコピーがハッシュテーブルにされ、そのがとしてされます。

インターンをして、==をしてをできるようにすることはです。しかし、これにはなががあります。[Pitfall - Interning](#)のをしてください。==をうことができます。ほとんどののはされません。

のの/の

Stringは、とのでをする2つのメソッドをします。

- すべてののをにするtoUpperCase
- すべてののをにするtoLowerCase

これらのメソッドののがにされたをすStringインスタンスをStringので、オブジェクトがされていないString Javaです。のについてはこれをてください [Javaにおけるの](#)

```
String string = "This is a Random String";
String upper = string.toUpperCase();
String lower = string.toLowerCase();

System.out.println(string);    // prints "This is a Random String"
System.out.println(lower);    // prints "this is a random string"
System.out.println(upper);    // prints "THIS IS A RANDOM STRING"
```

やなどののは、これらのメソッドのをけません。これらのメソッドは、のでのUnicodeをってうもあることにしてください。

これらのメソッドはロケールにしており、ロケールとはしてされるでされるとしないがじることがあります。としては、プログラミング、プロトコルキー、HTMLタグなどがありHTML。

たとえば、トルコロケールの"TITLE".toLowerCase()は" title "をします。ここで、ı (\u0131)はラテンのDOTLESS Iです。ロケールをしないにしたいをるには、Locale.ROOTをするとをするメソ

toLowerCase(Locale.ROOT) または toUpperCase(Locale.ROOT) に toUpperCase(Locale.ROOT) ます。

ほとんどの、Locale.ENGLISH を Locale.ENGLISH こともしいですが、ののは Locale.ROOT です。

なケーシングがなUnicodeのなリストは、[Unicode ConsortiumのWebサイト](#)にあります。

ASCIIののの/の

ASCIIのののとをするには、のアルゴリズムをできます。

ステップ

1. をします。
2. をします。
3. をにします。
4. するをします。
5. をにします。
6. つかったは、ががをします。
 - のは、のASCIIコードに32をします。
 - のは、のASCIIコードから32をいてください。
7. からのをします。
8. をにします。

Voila、キャラクターのケースがされました。

アルゴリズムのコードはのとおりで。

```
Scanner scanner = new Scanner(System.in);
System.out.println("Enter the String");
String s = scanner.next();
char[] a = s.toCharArray();
System.out.println("Enter the character you are looking for");
System.out.println(s);
String c = scanner.next();
char d = c.charAt(0);

for (int i = 0; i <= s.length(); i++) {
    if (a[i] == d) {
        if (d >= 'a' && d <= 'z') {
            d -= 32;
        } else if (d >= 'A' && d <= 'Z') {
            d += 32;
        }
        a[i] = d;
        break;
    }
}
s = String.valueOf(a);
System.out.println(s);
```

のでのの

のString `a` がString `b` にまれているかどうかをするには、`String.contains()` メソッドをのできます。

```
b.contains(a); // Return true if a is contained in b, false otherwise
```

`String.contains()` メソッドをして、`CharSequence` がString につかるかどうかをできます。このは、をする`a`で`b`とをしたで。

```
String str1 = "Hello World";
String str2 = "Hello";
String str3 = "helLO";

System.out.println(str1.contains(str2)); //prints true
System.out.println(str1.contains(str3)); //prints false
```

のライブデモ

String かのString でまるなをつけるには、`String.indexOf()` します。

```
String s = "this is a long sentence";
int i = s.indexOf('i'); // the first 'i' in String is at index 2
int j = s.indexOf("long"); // the index of the first occurrence of "long" in s is 10
int k = s.indexOf('z'); // k is -1 because 'z' was not found in String s
int h = s.indexOf("LoNg"); // h is -1 because "LoNg" was not found in String s
```

のライブデモ

`String.indexOf()` メソッドは、のString `char` またはString のインデックスをします。このメソッドは、つからないは`-1` します。

`String.indexOf()` メソッドではとがされます。

ケースをしたの

```
String str1 = "Hello World";
String str2 = "wOr";
str1.indexOf(str2); // -1
str1.toLowerCase().contains(str2.toLowerCase()); // true
str1.toLowerCase().indexOf(str2.toLowerCase()); // 6
```

のライブデモ

のさをする

String オブジェクトのさをするには、`length()` メソッドをびします。さは、のUTF-16コードのにしい。

```
String str = "Hello, World!";
System.out.println(str.length()); // Prints out 13
```

のライブデモ

Stringの`char`はUTF-16のです。が0x1000のUnicodeコードポイントたとえば、ほとんどのemojisは、2つのをします。コードポイントがUTF-16 `char`にまるかどうかにかかわらず、StringのUnicodeコードポイントのをカウントするには、`codePointCount`メソッドをできます。

```
int length = str.codePointCount(0, str.length());
```

Java 8ので、コードポイントのストリームをすることもできます。

```
int length = str.codePoints().count();
```

サブストリング

```
String s = "this is an example";
String a = s.substring(11); // a will hold the string starting at character 11 until the end
("example")
String b = s.substring(5, 10); // b will hold the string starting at character 5 and ending
right before character 10 ("is an")
String b = s.substring(5, b.length()-3); // b will hold the string starting at character 5
ending right before b' s lenght is out of 3 ("is an exam")
```

をスライスし、のにを/するために、をすることもできます。たとえば、をむのにしましたが、それをのとしてしたいとします。

```
String datestring = "2015-11-17"
datestring = datestring.substring(0, 4) + "-" + datestring.substring(5,7) + "-" +
datestring.substring(8,10);
//Result will be 2015-11-17
```

メソッドは、Stringをします。1つのパラメータがされた、パラメータはであり、ピースはStringまでされます。2つのパラメータがえられた、のパラメータはで、2のパラメータはののインデックスですインデックスのはまれません。なチェックは、のパラメーターを2のパラメーターからしてのさをするということです。

Java SE 7

JDK <7u6バージョンでは、`substring`メソッドは、のStringとじバッキング`char[]`をするStringをインスタンスし、のおよびさに`offset`および`count`フィールドをします。そのようなはメモリリークをきこすがあります。これは、`new String(s.substring(...))`をびしてコピーをにすることでぐことができます。その、`char[]`はガベージコレクションされます。

Java SE 7

JDK 7u6から、`substring`メソッドはに、の`char[]`をコピーして、のとしてさをにしますが、にメモリリークがないことをします。

のnのをする


```
String str = "My String";

System.out.println(str.charAt(0)); // "M"
System.out.println(str.charAt(1)); // "y"
System.out.println(str.charAt(2)); // " "
System.out.println(str.charAt(str.length-1)); // Last character "g"
```

の`n`のをするには、にびす `charAt(n)String`、 `n` あなたがしたいのインデックスであります

インデックス`n`は`0`からしているので、のは`n = 0`になります。

プラットフォームにしない

しいラインセパレータはプラットフォームからプラットフォームまでえばので、`\n` Unixライクなシステムまたはの`\r\n` Windowsでそれにアクセスするためのプラットフォームにしないをとっていることがしばしばです。Javaでは、システムプロパティからできます。

```
System.getProperty("line.separator")
```

Java SE 7

しいりがにになるので、のコードとまったくじをすショートカットメソッドのJava 7からできます。

```
System.lineSeparator()
```

しいりはプログラムのにされることはほとんどないので、になるたびにシステムプロパティからするのではなく、にすることをおめします。

`String.format`をする`%n`は、`\n`または`\r\n`ではなく`%n`して、プラットフォームにしないしいりをします。

```
System.out.println(String.format('line 1: %s.%nline 2: %s%n', lines[0], lines[1]));
```

カスタムオブジェクトの**`toString`**メソッドの

の`Person`クラスをしたとします。

```
public class Person {

    String name;
    int age;

    public Person (int age, String name) {
        this.age = age;
        this.name = name;
    }
}
```

しい `Person` オブジェクトをインスタンスするは、のようになります。

```
Person person = new Person(25, "John");
```

でコードでオブジェクトをするためにのステートメントをします。

```
System.out.println(person.toString());
```

のライブデモ

のようながられます。

```
Person@7ab89d
```

これは、 `Person` クラスのスーパークラスである `Object` クラスでされた `toString()` メソッドののです。
。 `Object.toString()` のドキュメントには、のようになっています。

`Object` クラスの `toString` メソッドは、オブジェクトがインスタンスであるクラスの、アットマーク `@`、およびオブジェクトのハッシュコードのなし16からなるをします。つまり、このメソッドはのにしいをします。

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

したがって、のあるをるには、 `toString()` メソッドをオーバーライドするがあります。

```
@Override
public String toString() {
    return "My name is " + this.name + " and my age is " + this.age;
}
```

これではのようになります。

```
My name is John and my age is 25
```

あなたもくことができます

```
System.out.println(person);
```

のライブデモ

、 `println()` にオブジェクトの `toString` メソッドをびします。

の

のりまたはに `String` をするには、のシグネチャをつ `String.split()` メソッドをします。

```
public String[] split(String regex)
```

またはのりは、のからされることにしてください。

りをする

```
String lineFromCsvFile = "Mickey;Bolton;12345;121216";
String[] dataCells = lineFromCsvFile.split(";");
// Result is dataCells = { "Mickey", "Bolton", "12345", "121216"};
```

をする

```
String lineFromInput = "What    do you need    from me?";
String[] words = lineFromInput.split("\\s+"); // one or more space chars
// Result is words = {"What", "do", "you", "need", "from", "me?"};
```

Stringリテラルをすることもできます

```
String[] firstNames = "Mickey, Frank, Alicia, Tom".split(", ");
// Result is firstNames = {"Mickey", "Frank", "Alicia", "Tom"};
```

パラメータはにとしてわれることをれないでください。

```
"aaa.bbb".split("."); // This returns an empty array
```

のでは、すべてのがりであるため、はのになります。

のメタであるりについてする

のは、でなメタとなされます

```
< > - = ! ( ) [ ] { } \ ^ $ | ? * + .
```

のデリミタのいずれかについてをするには、\\してエスケープするか、Pattern.quote() するがあります。

- Pattern.quote() をう

```
String s = "a|b|c";
String regex = Pattern.quote("|");
String[] arr = s.split(regex);
```

- をエスケープする

```
String s = "a|b|c";
String[] arr = s.split("\\|");
```

はのをする

デフォルトで `split(delimiter)` はのからののをします。このをオフにするには `split(delimiter, limit)` をのにして `split(delimiter, limit)` オーバーロード `split(delimiter, limit)` バージョンをするがあります。

```
String[] split = data.split("\\|", -1);
```

`split(regex)` は `split(regex, 0)` をにします。

`limit` パラメータは、パターンがされるをし、としてられるのさにします。

n が 0 よりきい、パターンはで $n - 1$ され、のさは n よりきくなく、ののエントリにはにしたりをえるすべてのがまれます。

n がの、パターンはできるだけくのされ、のさはです。

n が 0 の、パターンはできるだけくのされ、のさはで、ののはされます。

StringTokenizer

`split()` メソッドのほかに、 `StringTokenizer` をして `StringTokenizer` することもできます。

`StringTokenizer` は `String.split()` よりもさらにがあり、するのがしなくなっています。には、セット `String` としてえられるによってられたトークンをきすためにされています。はセパレータとしてします。こののため、 `String.split()` の2のさ `String.split()` 。

デフォルトのセットはです `\t\n\r\f`。のでは、を々にします。

```
String str = "the lazy fox jumped over the brown fence";
StringTokenizer tokenizer = new StringTokenizer(str);
while (tokenizer.hasMoreTokens()) {
    System.out.println(tokenizer.nextToken());
}
```

これはされます

```
the
lazy
fox
jumped
over
the
brown
fence
```

にはなるセットをできます。

```
String str = "jumped over";
// In this case character `u` and `e` will be used as delimiters
StringTokenizer tokenizer = new StringTokenizer(str, "ue");
while (tokenizer.hasMoreTokens()) {
```

```
System.out.println(tokenizer.nextToken());  
}
```

これはされます

```
j  
mp  
d ov  
r
```

りでをする

Java SE 8

ストリングのは、メソッド `String.join()` をしてできます。

```
String[] elements = { "foo", "bar", "foobar" };  
String singleString = String.join(" + ", elements);  
  
System.out.println(singleString); // Prints "foo + bar + foobar"
```

に、 `String.join()` はオーバーロードされた `String.join()` メソッドが `Iterable` ます。

をかくするには、 `StringJoiner` クラスをします。

```
StringJoiner sj = new StringJoiner(", ", "[", "]");  
// The last two arguments are optional,  
// they define prefix and suffix for the result string  
  
sj.add("foo");  
sj.add("bar");  
sj.add("foobar");  
  
System.out.println(sj); // Prints "[foo, bar, foobar]"
```

のストリームにするには、 `コレクタ` をします。

```
Stream<String> stringStream = Stream.of("foo", "bar", "foobar");  
String joined = stringStream.collect(Collectors.joining(", "));  
System.out.println(joined); // Prints "foo, bar, foobar"
```

とをここでするオプションもあります

```
Stream<String> stringStream = Stream.of("foo", "bar", "foobar");  
String joined = stringStream.collect(Collectors.joining(", ", "{", "}"));  
System.out.println(joined); // Prints "{foo, bar, foobar}"
```

の

をにしてきにすることができます。

1. StringBuilder / StringBuffer

```
String code = "code";
System.out.println(code);

StringBuilder sb = new StringBuilder(code);
code = sb.reverse().toString();

System.out.println(code);
```

```
2. String code = "code";
System.out.println(code);

char[] array = code.toCharArray();
for (int index = 0, mirroredIndex = array.length - 1; index < mirroredIndex; index++, mirroredIndex--) {
    char temp = array[index];
    array[index] = array[mirroredIndex];
    array[mirroredIndex] = temp;
}

// print reversed
System.out.println(new String(array));
```

のまたはのをえる

[org.apache.commons.lang3.StringUtils](https://commons.apache.org/lang3/StringUtils/)の `countMatches` メソッドは、`String` のまたはのをカウントするためにされます。

```
import org.apache.commons.lang3.StringUtils;

String text = "One fish, two fish, red fish, blue fish";

// count occurrences of a substring
String stringTarget = "fish";
int stringOccurrences = StringUtils.countMatches(text, stringTarget); // 4

// count occurrences of a char
char charTarget = ',';
int charOccurrences = StringUtils.countMatches(text, charTarget); // 3
```

それのは、Java APIでもをできます。

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

String text = "One fish, two fish, red fish, blue fish";
System.out.println(countStringInString("fish", text)); // prints 4
System.out.println(countStringInString(",", text)); // prints 3

public static int countStringInString(String search, String text) {
    Pattern pattern = Pattern.compile(search);
```

```

    Matcher matcher = pattern.matcher(text);

    int stringOccurrences = 0;
    while (matcher.find()) {
        stringOccurrences++;
    }
    return stringOccurrences;
}

```

とStringBuilders

のは、`+`をしてできます。えば

```

String s1 = "a";
String s2 = "b";
String s3 = "c";
String s = s1 + s2 + s3; // abc

```

、コンパイラでは、`StringBuilder`むメソッドをしてのをします。コンパイルすると、コードはのようになります。

```

StringBuilder sb = new StringBuilder("a");
String s = sb.append("b").append("c").toString();

```

`StringBuilder`は、さまざまなをするためのオーバーロードされたメソッドがいくつかありますたとえば、`String`わりに`int`をするなど。えば、はのようになります。

```

String s1 = "a";
String s2 = "b";
String s = s1 + s2 + 2; // ab2

```

のようにします。

```

StringBuilder sb = new StringBuilder("a");
String s = sb.append("b").append(2).toString();

```

のは、コードののでにされるなをしています。には、`StringBuilder`インスタンスがまれます。いくつかの、はループのようににされます。

```

String result = "";
for(int i = 0; i < array.length; i++) {
    result += extractElement(array[i]);
}
return result;

```

このような、コンパイラのはされず、でしい`StringBuilder`オブジェクトがされます。これは、の`StringBuilder`をするようにコードをにすることによってできます。

```

StringBuilder result = new StringBuilder();
for(int i = 0; i < array.length; i++) {

```

```

        result.append(extractElement(array[i]));
    }
    return result.toString();

```

`StringBuilder`は、16のでされます。よりきなをすることがにかっているは、あらかじめなサイズでしてバッファのサイズをするはありません。

```

StringBuilder buf = new StringBuilder(30); // Default is 16 characters
buf.append("0123456789");
buf.append("0123456789"); // Would cause a reallocation of the internal buffer otherwise
String result = buf.toString(); // Produces a 20-chars copy of the string

```

くのをするは、`StringBuilder`をすることをおめします。

```

StringBuilder buf = new StringBuilder(100);
for (int i = 0; i < 100; i++) {
    buf.setLength(0); // Empty buffer
    buf.append("This is line ").append(i).append('\n');
    outfile.write(buf.toString());
}

```

のスレッドがじバッファにきんでいるのみ、[StringBuffer](#)をします。これは`StringBuilder` `synchronized`バージョンです。しかし、は1つのスレッドだけがバッファにきむため、はせずに`StringBuilder`をするがです。

concat メソッドの

```

String string1 = "Hello ";
String string2 = "world";
String string3 = string1.concat(string2); // "Hello world"

```

これは、に`string2`がされた`string1`というしいをします。のように、`concat`メソッドをリテラルとともにすることもできます。

```

"My name is ".concat("Buyya");

```

ストリングのをきえる

きえる2つのまたは。

のStringオブジェクトはされず、りはされたStringをします。

に

のをののにきえます。

```

String replace(char oldChar, char newChar)

```


こののすべてのoldCharをnewCharにきえたのしいをします。

```
String s = "popcorn";
System.out.println(s.replace('p', 'W'));
```

```
WoWcorn
```

のびをののびにきえます。

```
String replace(CharSequence target, CharSequence replacement)
```

リテラルターゲットシーケンスとするこののを、されたリテラルシーケンスにきえます。

```
String s = "metal petal et al.";
System.out.println(s.replace("etal", "etallica"));
```

```
metallica petallica et al.
```

グループは、\$1のようにグループをするために\$をします。

すべてのをする

```
String replaceAll(String regex, String replacement)
```

されたにするこののを、されたにきえます。

```
String s = "spiral metal petal et al.";
System.out.println(s.replaceAll("(\\w*etal)", "$1lica"));
```

```
spiral metallica petallica et al.
```

のみをする

```
String replaceFirst(String regex, String replacement)
```

されたにするこののを、されたできえます。

```
String s = "spiral metal petal et al.";
System.out.println(s.replaceAll("(\\w*etal)", "$1lica"));
```

```
spiral metallica petal et al.
```

のとのをする

`trim()` メソッドは、`String` をします。

```
String s = new String("  Hello World!! ");
String t = s.trim(); // t = "Hello World!!"
```

するをたないを `trim` すると、`String` インスタンスがされます。

`trim()` メソッドには、`Character.isWhitespace()` メソッドでされているとはなるのがあります。

- コード `U+0000 U+0020` すべてのASCIIはとなされ、`trim()` によってされ `trim()`。これには `U+0020` 'SPACE'、`U+0009` 'CHARACTER TABULATION'、`U+000A` 'LINE FEED' `U+0009` 'CHARACTER TABULATION' `U+000A` 'LINE FEED'、`U+000D` 'CARRIAGE RETURN' キャラクターだけでなく、`U+0007` 'BELL' などのキャラクターもまれます。
- `U+00A0` 'NO-BREAK SPACE' や `U+2003` 'EM SPACE' ようなUnicodeは `trim()` によってされません。

String プールとヒープストレージ

くのJavaオブジェクトとに、すべての `String` インスタンスはリテラルでもヒープにされます。JVMがつかったとき `String` ヒープにはのをたないリテラルを、JVMは、する `String` ヒープにインスタンスをし、それはまた、たにされたする `String` プールのインスタンス。`String` リテラルへののは、ヒープのにされた `String` インスタンスにきえられます。

のをてみましょう

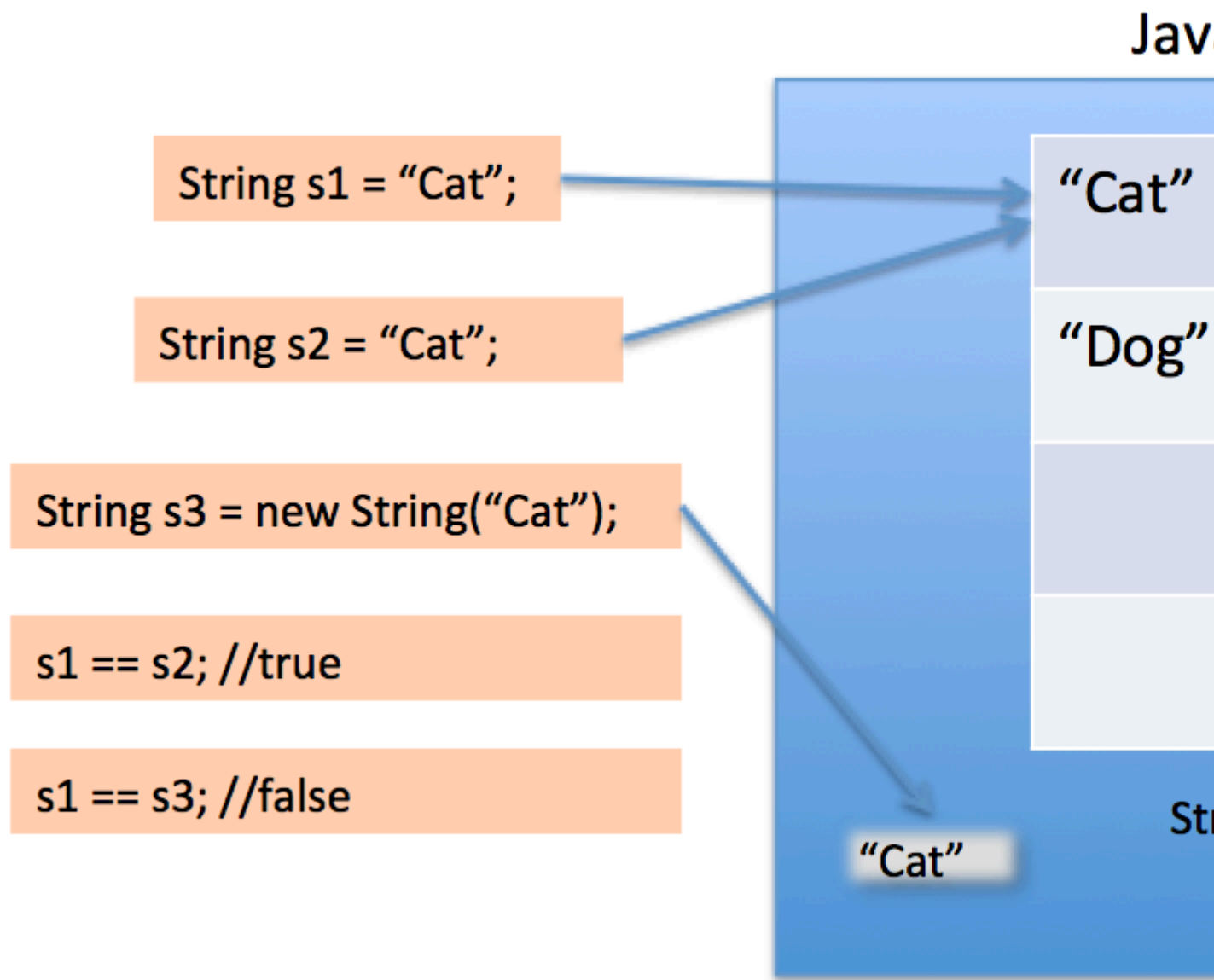
```
class Strings
{
    public static void main (String[] args)
    {
        String a = "alpha";
        String b = "alpha";
        String c = new String("alpha");

        //All three strings are equivalent
        System.out.println(a.equals(b) && b.equals(c));

        //Although only a and b reference the same heap object
        System.out.println(a == b);
        System.out.println(a != c);
        System.out.println(b != c);
    }
}
```

のはのとおりで。

```
true
true
true
true
```



をしてをすると、StringプールのじをつStringがにされます。つかったは、をします。プールにし
いStringをしてからをします。

しかし、newをうと、StringクラスはヒープにしいStringオブジェクトをします。internメソッド
をもってプールにれたり、じをつプールからのStringオブジェクトをしたりすることができます。

Stringプールもヒープにされます。

Java SE 7

Java 7よりでは、String リテラルはサイズのPermGenのメソッドのランタイムプールにされていま
した。

StringプールもPermGenいました。

Java SE 7

JDK 7では、インターネットされたは、なJavaヒープではりてられなくなりましたが、アプリケーションによってされたのオブジェクトとともに、Javaヒープのいおよびい。このにより、メインJavaヒープにくのデータがされ、なではデータがなくなるため、ヒープサイズのがになることがあります。ほとんどのアプリケーションでは、このによりヒープのはごくわずかですが、くのクラスをみんだり、`String.intern()`メソッドをにしているアプリケーションがいほど、きないがられます。

ケーススイッチ

Java SE 7

`switch`はをしないようにパラメータすることはできませんが、になは、`toLowerCase()`または`toUpperCase`をしてにしてなをすることができます。

```
switch (myString.toLowerCase()) {
    case "case1" :
        ...
        break;
    case "case2" :
        ...
        break;
}
```

してください

- `Locale`は、[するケースが](#)どのように[こる](#)かにします。
- ラベルにをまないようにをつけなければなりません。されことはありません

オンラインでをむ <https://riptutorial.com/ja/java/topic/109/>

164: トークナイザ

き

java.util.StringTokenizerクラスをすると、をトークンにできます。それはをすなです。

りのセットトークンをするは、またはトークンごとにすることができます。

Examples

StringTokenizer スペースで

```
import java.util.StringTokenizer;
public class Simple{
    public static void main(String args[]){
        StringTokenizer st = new StringTokenizer("apple ball cat dog"," ");
        while (st.hasMoreTokens()) {
            System.out.println(st.nextToken());
        }
    }
}
```

ネコ

StringTokenizer コンマで ','

```
public static void main(String args[]) {
    StringTokenizer st = new StringTokenizer("apple,ball cat,dog", ",");
    while (st.hasMoreTokens()) {
        System.out.println(st.nextToken());
    }
}
```

ボールキャット

オンラインでトークナイザをむ <https://riptutorial.com/ja/java/topic/10563/> トークナイザ

165: との

Examples

のデータを**String**にする

- StringクラスのvalueOfメソッドをして、のプリミティブデータのをStringとしてできます。

えば

```
int i = 42;
String string = String.valueOf(i);
//string now equals "42".
```

このメソッドは、 float 、 double 、 boolean 、 およびObjectなどのデータのにしてもオーバーロードされます。

- .toStringをびすことで、のObjectクラスのインスタンスをStringとしてすることもできます。これがなをえるためには、クラスはtoString()オーバーライドするがあります。Javaライブラリクラスのほとんどは、 Date などです。

えば

```
Foo foo = new Foo(); //Any class.
String stringifiedFoo = foo.toString().
```

ここでstringifiedFooにはfooがStringとしてまれています。

また、のをのよういStringにすることもできます。

```
int i = 10;
String str = i + "";
```

それともなです

```
String str = 10 + "";
```

バイトへの/バイトからの

をバイトにエンコードするには、 String.getBytes() メソッドをします。のセットのいずれかをJavaランタイムでできます。

```
byte[] bytes = "test".getBytes(StandardCharsets.UTF_8);
```

デコードする

```
String testString = new String(bytes, StandardCharsets.UTF_8);
```

なインポートをすると、コールをさらにできます。

```
import static java.nio.charset.StandardCharsets.UTF_8;
...
byte[] bytes = "test".getBytes(UTF_8);
```

あまりでないセットのは、セットをでできます。

```
byte[] bytes = "test".getBytes("UTF-8");
```

その

```
String testString = new String (bytes, "UTF-8");
```

これは、チェックされた`UnsupportedCharsetException`をするがあることをします。

のびしでは、デフォルトのセットがされます。デフォルトのセットはプラットフォームであり、Windows、Mac、およびLinuxプラットフォームではになります。

```
byte[] bytes = "test".getBytes();
```

その

```
String testString = new String(bytes);
```

なとバイトは、これらのメソッドでまたはスキップすることができます。をするためのインスタンスのための-よりくののためにあなたがすることをしている`CharsetEncoder`と`CharsetDecoder`クラスを。

Base64エンコーディング/デコード

バイナリデータを`base64`でエンコードしたとしてエンコードするがあります。

このために、`javax.xml.bind`パッケージの`DatatypeConverter`クラスをできます。

```
import javax.xml.bind.DatatypeConverter;
import java.util.Arrays;

// arbitrary binary data specified as a byte array
byte[] binaryData = "some arbitrary data".getBytes("UTF-8");

// convert the binary data to the base64-encoded string
String encodedData = DatatypeConverter.printBase64Binary(binaryData);
// encodedData is now "c29tZSBhcmJpdHJhcnkgZGF0YQ=="
```

```
// convert the base64-encoded string back to a byte array
byte[] decodedData = DatatypeConverter.parseBase64Binary(encodedData);

// assert that the original data and the decoded data are equal
assert Arrays.equals(binaryData, decodedData);
```

Apache commons-codec

あるいは、[Apache commons-codec](#)の `Base64` をすることもできます。

```
import org.apache.commons.codec.binary.Base64;

// your blob of binary as a byte array
byte[] blob = "someBinaryData".getBytes();

// use the Base64 class to encode
String binaryAsAString = Base64.encodeBase64String(blob);

// use the Base64 class to decode
byte[] blob2 = Base64.decodeBase64(binaryAsAString);

// assert that the two blobs are equal
System.out.println("Equal : " + Boolean.toString(Arrays.equals(blob, blob2)));
```

このプログラムをすると、`someBinaryData` がにしやすい `UTF-8 String` オブジェクトである `c29tZUJpbmFyeURhdGE=` にエンコードされていることがわかります。

Java SE 8

そのは `Base64` でできます

```
// encode with padding
String encoded = Base64.getEncoder().encodeToString(someByteArray);

// encode without padding
String encoded = Base64.getEncoder().withoutPadding().encodeToString(someByteArray);

// decode a String
byte [] barr = Base64.getDecoder().decode(encoded);
```

をにする

をプリミティブなまたはのラッパーにします。

ラッパークラスは、`String` をするプリミティブにする `parseXxx` メソッドをします。のコードは、`Integer.parseInt` メソッドをして `String` を `int` し `Integer.parseInt` 。

```
String string = "59";
int primitive = Integer.parseInt(string);
```


ラッパークラスのインスタンスに `String` にするには、ラッパークラス `valueOf` メソッドのオーバーロードをするか、

```
String string = "59";
Integer wrapper = Integer.valueOf(string);
```

またはボクシングJava 5にしている

```
String string = "59";
Integer wrapper = Integer.parseInt(string); // 'int' result is autoboxed
```

のパターンは、`byte`、`short`、`int`、`long`、`float` および `double`、およびするラッパークラス `Byte`、`Short`、`Integer`、`Long`、`Float` および `Double` でします。

をしてをにする

```
String integerAsString = "0101"; // binary representation
int parseInt = Integer.parseInt(integerAsString, 2);
Integer valueOfInteger = Integer.valueOf(integerAsString, 2);
System.out.println(valueOfInteger); // prints 5
System.out.println(parseInt); // prints 5
```

されないでないか、のをすにして、の `valueOf(String)` または `parseXxx(...)` メソッドがびされた、チェックされていない `NumberFormatException` がスローされます。

‘`InputStream`’から‘`String`’をる

`String` は、バイトコンストラクタをして `InputStream` からみることができます。

```
import java.io.*;

public String readString(InputStream input) throws IOException {
    byte[] bytes = new byte[50]; // supply the length of the string in bytes here
    input.read(bytes);
    return new String(bytes);
}
```

これは、システムのデフォルトのセットをしますが、のセットをすることもできます。

```
return new String(bytes, Charset.forName("UTF-8"));
```

をのデータにする

をのようにさまざまなJavaにできます。

`int`から`int`への

```
String number = "12";
int num = Integer.parseInt(number);
```

する

```
String number = "12.0";
float num = Float.parseFloat(number);
```

doubleにする

```
String double = "1.47";
double num = Double.parseDouble(double);
```

ブールへの

```
String falseString = "False";
boolean falseBool = Boolean.parseBoolean(falseString);    // falseBool = false

String trueString = "True";
boolean trueBool = Boolean.parseBoolean(trueString);       // trueBool = true
```

をくする

```
String number = "47";
long num = Long.parseLong(number);
```

BigIntegerへの

```
String bigNumber = "21";
BigInteger reallyBig = new BigInteger(bigNumber);
```

を**BigDecimal**にする

```
String bigFraction = "17.21455";
BigDecimal reallyBig = new BigDecimal(bigFraction);
```

の

のは、にされたではないをしようとすると、チェックされていない `NumberFormatException` スローし `NumberFormatException`。 「 」トピックでは、そのようなをするについてします。

をできることをテストするは、のように `tryParse...` メソッドをできます。

```
boolean tryParseInt (String value) {
    try {
        Integer.parseInt(value);
        return true;
    } catch (NumberFormatException e) {
        return false;
    }
}
```

しかし、この `tryParse...` メソッドをするにびすことは、いなくなです。 `parse...` メソッドをびし

てを`parse...`がいでしょう。

オンラインでとのをむ <https://riptutorial.com/ja/java/topic/6678/>との

166: をにする

ここでのコンテンツをうことではないため、はをしてはいけません。むしろのターゲットのののターゲットののののしなければなりません。えは8のの、ののをげるつまりさせるがあります。

a b c d e f g h i j k l m n o p q r s t u v w x y z

^ ^ ^

ののにするがあったのスペースをします。

Examples

をのさのにする

トリックは、`\G`を使ってルック・バックをうことです。これは"のマッチのわり"をします

```
String[] parts = str.split("(?<=\\G.{8})");
```

は、のマッチのに8にマッチします。この、マッチはゼロであるため、よりに「のマッチのに8」
とすることができます。

なことに、 $\backslash_G$ はのにされるので、ののにしてもします。

をのにする

のさのと同じですが、lengthをregexにします

```
int length = 5;
String[] parts = str.split("(?<=\\G.{ " + length + "})");
```

オンラインでをにするをむ [https://riptutorial.com/ja/java/topic/5613/](https://riptutorial.com/ja/java/topic/5613)をにする

167: しいファイルI/O

- `Paths.getString first、String ... more`// `String`で`Path`インスタンスをします。
- `Paths.getURI uri`// `URI`でパスインスタンスをします。

Examples

パスの

`Path`クラスは、ファイルシステムのパスをプログラマティカルにするためにされますしたがって、しないファイルであってもファイルとディレクトリをすることができます

パスは、ヘルパークラス `Paths` をしてできます。

```
Path p1 = Paths.get("/var/www");
Path p2 = Paths.get(URI.create("file:///home/testuser/File.txt"));
Path p3 = Paths.get("C:\\Users\\DentAr\\Documents\\HHGTDG.odt");
Path p4 = Paths.get("/home", "arthur", "files", "diary.tex");
```

パスにするの

パスにするは、`Path`オブジェクトのメソッドをしてできます。

- `toString()` は、パスのをします。

```
Path p1 = Paths.get("/var/www"); // p1.toString() returns "/var/www"
```

- `getFileName()` は、ファイルまたは、には、パスのの

```
Path p1 = Paths.get("/var/www"); // p1.getFileName() returns "www"
Path p3 = Paths.get("C:\\Users\\DentAr\\Documents\\HHGTDG.odt"); // p3.getFileName()
returns "HHGTDG.odt"
```

- `getNameCount()` は、パスをするのをします。

```
Path p1 = Paths.get("/var/www"); // p1.getNameCount() returns 2
```

- `getName(int index)` れたインデックスにあるをします。

```
Path p1 = Paths.get("/var/www"); // p1.getName(0) returns "var", p1.getName(1) returns
"www"
```

- `getParent()` は、ディレクトリのパスをします。

```
Path p1 = Paths.get("/var/www"); // p1.getParent().toString() returns "/var"
```

- `getRoot()` はパスのルートをしします。

```
Path p1 = Paths.get("/var/www"); // p1.getRoot().toString() returns "/"
Path p3 = Paths.get("C:\\Users\\DentAr\\Documents\\HHGTDG.odt"); //
p3.getRoot().toString() returns "C:\\"
```

パスの

2つのパスにする

パスは `resolve()` メソッドをしてできます。されるパスは、ルートをまないパスであるパスでなければなりません。

```
Path p5 = Paths.get("/home/");
Path p6 = Paths.get("arthur/files");
Path joined = p5.resolve(p6);
Path otherJoined = p5.resolve("ford/files");
```

```
joined.toString() == "/home/arthur/files"
otherJoined.toString() == "/home/ford/files"
```

パスの

パスにはをめることができます。あなたがっているディレクトリをしていますと..`ディレクトリ`をしています。

パスでされる、`パス`のをすることなく、いつでもすることができ、..`のとにすることができ`ます。

`Paths API`では、`.normalize()`メソッドをしてこれをい`.normalize()`。

```
Path p7 = Paths.get("/home/./arthur/../ford/files");
Path p8 = Paths.get("C:\\Users\\..\\..\\Program Files");
```

```
p7.normalize().toString() == "/home/ford/files"
p8.normalize().toString() == "C:\\Program Files"
```

ファイルシステムをったの

ファイルシステムとするには、クラス`Files`メソッドをし`Files`。

をする

パスがファイルまたはディレクトリのあるものにするには、`Files.exists(Path path)` を使います。

```
Files.exists(Path path)
```

そして

```
Files.notExists(Path path)
```

`!Files.exists(path)` **neccesarily**にしくなるようにするはありません `Files.notExists(path)`、つのなシナリオがあるので

- ファイルまたはディレクトリがある `exists` は `true` をし、 `notExists` はこの `false` をします
- ファイルまたはディレクトリがない `exists` と `false` をし、 `notExists` が `true` する
- ファイルやディレクトリにはできませんアクセスなど `exists` と `notExists` `exists` `false` をします。

パスがファイルまたはディレクトリをしているかどうかをする

これは、 `Files.isRegularFile(Path path)` `Files.isDirectory(Path path)` と `Files.isRegularFile(Path path)` `Files.isDirectory(Path path)` をして `Files.isRegularFile(Path path)`

```
Path p1 = Paths.get("/var/www");
Path p2 = Paths.get("/home/testuser/File.txt");
```

```
Files.isDirectory(p1) == true
Files.isRegularFile(p1) == false

Files.isDirectory(p2) == false
Files.isRegularFile(p2) == true
```

プロパティの

これは、`Files.isReadable(Path path)`、`Files.isWritable(Path path)`、`Files.isExecutable(Path path)`、`Files.isHidden(Path path)`、`Files.isSymbolicLink(Path path)` を使えます。

```
Files.isReadable(Path path)
Files.isWritable(Path path)
Files.isExecutable(Path path)

Files.isHidden(Path path)
Files.isSymbolicLink(Path path)
```

MIMEタイプの

```
Files.probeContentType(Path path)
```

これは、ファイルのMIMEタイプをしようします。のように、MIMEタイプStringをします。

- text/plain ファイルのtext/plain
- HTMLページのtext/html
- PDFファイルのためのapplication/pdf
- PNGファイルのimage/png

ファイルのみみ

ファイルは、Filesクラスをしてバイトおよびでみることができます。

```
Path p2 = Paths.get(URI.create("file:///home/testuser/File.txt"));
byte[] content = Files.readAllBytes(p2);
List<String> linesOfContent = Files.readAllLines(p2);
```

Files.readAllLines() オプションでパラメータとしてcharsetをとりますデフォルトはStandardCharsets.UTF_8。

```
List<String> linesOfContent = Files.readAllLines(p2, StandardCharsets.ISO_8859_1);
```

ファイルのきみ

ファイルは、Filesクラスをしてバイトとラインでくことができます

```
Path p2 = Paths.get("/home/testuser/File.txt");
List<String> lines = Arrays.asList(
    new String[]{"First line", "Second line", "Third line"});

Files.write(p2, lines);
```

```
Files.write(Path path, byte[] bytes)
```

のファイルがきされ、しないファイルがされます。

オンラインでしいファイルI/Oをむ <https://riptutorial.com/ja/java/topic/5519/しいファイルi-o>

168: クラス

- `Date object = new Date();`
- `Date object = new Date(long date);`

パラメーター

パラメータ	
パラメータなし	りてもいいミリをしていいDateオブジェクトをします。
い	"The epoch"197011、00:00:00 GMTのミリにされたでいいDateオブジェクトをします。

には、Java Dateオブジェクトはlongとしてされます。のエポックと呼ばれるからのミリです。のJava Dateクラスにはタイムゾーンなどをうメソッドがありましたが、これらはいいCalendarクラスのに変わりました。

したがって、あなたのコードでやりたいことがのをしているは、Dateクラスをしてすることが出来ます。ただし、ができるバージョンのをすることは、Calendarクラスをします、、、などをするには、のをします。のミリは、なるでなるとしてされます。は "ローカル"のタイムゾーンにするが、フォーマットでは、のいくつかのタイムゾーンでするがあることをするがあります。

また、JVMでされるクロックのはミリではないことにしてください。クロックは10ミリごとに「ティック」するだけなので、タイミングをとるとそのレベルでにをすることになることはできません。

インポートステートメント

```
import java.util.Date;
```

Dateクラスは、`java.util`パッケージからインポートできます。

あぶない

Dateインスタンスはであるため、それらをするときスレッドセーフなコードを書くことがになり、ってへのきみアクセスをするがあります。たとえば、のクラスで`getDate()`メソッドをすると、びしはトランザクションのをできます。

```
public final class Transaction {  
    private final Date date;  
  
    public Date getTransactionDate() {  
        return date;  
    }  
}
```

```
}  
}
```

は、`date` フィールドのコピーをすか、Java 8でされた`java.time`のしいAPIをすることです。

`Date`クラスのほとんどのコンストラクタメソッドはされており、すべきではありません。ほとんどの、には`Calendar`クラスをすることをおめします。

Java 8

Javaの8は、パッケージにしいとのAPIをし`java.time`め、`LOCALDATE`とに`LocalTime`。

`java.time`パッケージのクラスは、いやすくなったオーバーホールされたAPIをします。Java 8にきむは、このしいAPIをすることをくおめします。と`java.time`。*をしてください。

Examples

Dateオブジェクトの

```
Date date = new Date();  
System.out.println(date); // Thu Feb 25 05:03:59 IST 2016
```

この`Date`オブジェクトには、このオブジェクトがされたのとがまれます。

```
Calendar calendar = Calendar.getInstance();  
calendar.set(90, Calendar.DECEMBER, 11);  
Date myBirthDate = calendar.getTime();  
System.out.println(myBirthDate); // Mon Dec 31 00:00:00 IST 1990
```

`Date`オブジェクトは、データコンストラクタのがでされていないため、`Calendar`インスタンスをじてするのがです。これをうには、`Factory`メソッドから`Calendar`クラスのインスタンスをすることがあります。に、をして、、をできます。また、カレンダークラスでされるのは、みやすさをさせ、エラーをらすことができます。

```
calendar.set(90, Calendar.DECEMBER, 11, 8, 32, 35);  
Date myBirthDatenTime = calendar.getTime();  
System.out.println(myBirthDatenTime); // Mon Dec 31 08:32:35 IST 1990
```

とともに、、、のにをすこともできます。

Dateオブジェクトの

カレンダー、、ローカル

Java SE 8

before、after、compareToおよびequalsメソッド

```
//Use of Calendar and Date objects
final Date today = new Date();
final Calendar calendar = Calendar.getInstance();
calendar.set(1990, Calendar.NOVEMBER, 1, 0, 0, 0);
Date birthdate = calendar.getTime();

final Calendar calendar2 = Calendar.getInstance();
calendar2.set(1990, Calendar.NOVEMBER, 1, 0, 0, 0);
Date samebirthdate = calendar2.getTime();

//Before example
System.out.printf("Is %1$tF before %2$tF? %3$b%n", today, birthdate,
Boolean.valueOf(today.before(birthdate)));
System.out.printf("Is %1$tF before %1$tF? %3$b%n", today, today,
Boolean.valueOf(today.before(today)));
System.out.printf("Is %2$tF before %1$tF? %3$b%n", today, birthdate,
Boolean.valueOf(birthdate.before(today)));

//After example
System.out.printf("Is %1$tF after %2$tF? %3$b%n", today, birthdate,
Boolean.valueOf(today.after(birthdate)));
System.out.printf("Is %1$tF after %1$tF? %3$b%n", today, birthdate,
Boolean.valueOf(today.after(today)));
System.out.printf("Is %2$tF after %1$tF? %3$b%n", today, birthdate,
Boolean.valueOf(birthdate.after(today)));

//Compare example
System.out.printf("Compare %1$tF to %2$tF: %3$d%n", today, birthdate,
Integer.valueOf(today.compareTo(birthdate)));
System.out.printf("Compare %1$tF to %1$tF: %3$d%n", today, birthdate,
Integer.valueOf(today.compareTo(today)));
System.out.printf("Compare %2$tF to %1$tF: %3$d%n", today, birthdate,
Integer.valueOf(birthdate.compareTo(today)));

//Equal example
System.out.printf("Is %1$tF equal to %2$tF? %3$b%n", today, birthdate,
Boolean.valueOf(today.equals(birthdate)));
System.out.printf("Is %1$tF equal to %2$tF? %3$b%n", birthdate, samebirthdate,
Boolean.valueOf(birthdate.equals(samebirthdate)));
System.out.printf(
    "Because birthdate.getTime() -> %1$d is different from samebirthdate.getTime() ->
    %2$d, there are milliseconds!\n",
    Long.valueOf(birthdate.getTime()), Long.valueOf(samebirthdate.getTime()));

//Clear ms from calendars
calendar.clear(Calendar.MILLISECOND);
calendar2.clear(Calendar.MILLISECOND);
birthdate = calendar.getTime();
samebirthdate = calendar2.getTime();

System.out.printf("Is %1$tF equal to %2$tF after clearing ms? %3$b%n", birthdate,
samebirthdate,
Boolean.valueOf(birthdate.equals(samebirthdate)));
```

isBefore、isAfter、compareToおよびequalsメソッド

```
//Use of LocalDate
final LocalDate now = LocalDate.now();
final LocalDate birthdate2 = LocalDate.of(2012, 6, 30);
final LocalDate birthdate3 = LocalDate.of(2012, 6, 30);

//Hours, minutes, second and nanoOfsecond can also be configured with an other class
LocalDateTime
//LocalDateTime.of(year, month, dayOfMonth, hour, minute, second, nanoOfSecond);

//isBefore example
System.out.printf("Is %1$tF before %2$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isBefore(birthdate2)));
System.out.printf("Is %1$tF before %1$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isBefore(now)));
System.out.printf("Is %2$tF before %1$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(birthdate2.isBefore(now)));

//isAfter example
System.out.printf("Is %1$tF after %2$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isAfter(birthdate2)));
System.out.printf("Is %1$tF after %1$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isAfter(now)));
System.out.printf("Is %2$tF after %1$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(birthdate2.isAfter(now)));

//compareTo example
System.out.printf("Compare %1$tF to %2$tF %3$d%n", now, birthdate2,
Integer.valueOf(now.compareTo(birthdate2)));
System.out.printf("Compare %1$tF to %1$tF %3$d%n", now, birthdate2,
Integer.valueOf(now.compareTo(now)));
System.out.printf("Compare %2$tF to %1$tF %3$d%n", now, birthdate2,
Integer.valueOf(birthdate2.compareTo(now)));

//equals example
System.out.printf("Is %1$tF equal to %2$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.equals(birthdate2)));
System.out.printf("Is %1$tF to %2$tF? %3$b%n", birthdate2, birthdate3,
Boolean.valueOf(birthdate2.equals(birthdate3)));

//isEqual example
System.out.printf("Is %1$tF equal to %2$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isEqual(birthdate2)));
System.out.printf("Is %1$tF to %2$tF? %3$b%n", birthdate2, birthdate3,
Boolean.valueOf(birthdate2.isEqual(birthdate3)));
```

Java 8よりの

Javaの8のに、をしてすることができ[java.util.Calendar](#)のと[java.util.Date](#)クラスを。クラスはをす

る4つのメソッドをします

-
-
- [compareToanotherDate](#)
- [equalsObject obj](#)

after、before、compareToおよびequalsメソッドは、の[getTime](#)メソッドによってされたをします。

compareToメソッドはのをします。

- 0より大きいDateがDateのにある
- 0より小さいDateがDateのにある
- が0のDateがDateとしい

equalsにすように、にえられていないのは、ミリのよう、じでされていがないため、はくべきであることができます。

Java 8

Java 8では、DateをするしいObjectは[java.time.LocalDate](#)でできます。 `LocalDate` `ChronoLocalDate`、またはカレンダーシステムは、プラグインであるの。

のをるためには、Object [java.time.LocalDateTime](#)をするがあります。 `LocalDate`と`LocalDateTime`はじメソッドをにします。

`LocalDate`をしてをすることは、`LocalDate`をする`LocalDate`とはなり`ChronoLocalDate`なぜなら、やカレンダーシステムはのものをにれていないからです。

ほとんどのアプリケーションで`LocalDate`をするがあるため、`ChronoLocalDate`にははまれていません。 [ここでさらにむ](#)。

ほとんどのアプリケーションでは、この[`ChronoLocalDate`]インターフェイスではなく、メソッドシグネチャ、フィールドおよびを`LocalDate`としてするがあります。

`LocalDate`は、をする5つのメソッドがあります。

- [isAfterChronoLocalDateその](#)
- [isBeforeChronoLocalDateその](#)
- [isEqualChronoLocalDateその](#)
- [compareToChronoLocalDateその](#)
- [equalsObject obj](#)

`LocalDate`パラメータの、`isAfter`、`isBefore`、`isEqual`、`equals`、および`compareTo`は、このメソッドをします。

```
int compareTo(LocalDate otherDate) {
    int cmp = (year - otherDate.year);
    if (cmp == 0) {
        cmp = (month - otherDate.month);
        if (cmp == 0) {
            cmp = (day - otherDate.day);
        }
    }
    return cmp;
}
```

`isEqualcompareTo`をびすのにし、パラメータがににしい`equals`メソッドのチェックに`equals`。

`ChronoLocalDate`のクラスのインスタンスの`ChronoLocalDate`、は`Epoch Day`をしてされます。エポックは、0が1970-01-01ISOのなカウントです。

をのにする

`SimpleDateFormat`クラスの`format()`は、されたパターンをして、`Date`オブジェクトをのフォーマットの`String`オブジェクトにするのにちます。

```
Date today = new Date();

SimpleDateFormat dateFormat = new SimpleDateFormat("dd-MMM-yy"); //pattern is specified here
System.out.println(dateFormat.format(today)); //25-Feb-16
```

`applyPattern()`をすると、パターンをできます。

```
dateFormat.applyPattern("dd-MM-yyyy");
System.out.println(dateFormat.format(today)); //25-02-2016

dateFormat.applyPattern("dd-MM-yyyy HH:mm:ss E");
System.out.println(dateFormat.format(today)); //25-02-2016 06:14:33 Thu
```

ここで、`mm`の`m`はをし、`MM`の`M`はをします。の`"y"`_yはをし、の`"Y"`_Yは"のをします。

をにする

`SimpleDateFormat`クラスの`parse()`は、`String`パターンを`Date`オブジェクトにするのにちます。

```
DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.SHORT, Locale.US);
String dateStr = "02/25/2016"; // input String
Date date = dateFormat.parse(dateStr);
System.out.println(date.getYear()); // 116
```

テキストには4つのなるスタイルがあり、`SHORT`、`MEDIUM`これがデフォルト、`LONG`と`FULL`すべてロケールにします。ロケールがされていない、システムのデフォルトロケールがされます。

スタイル	Locale.US	Locale.France
シヨート	6/30/09	30/06/09
	2009630	2009630
いです	2009630	2009630
FULL	2009630	mardi 2009630

な

yyyy/MM/dd hh:mm:ss のでのコードをすると、のがされます。

2016/04/19 1145.36

```
// define the format to use
String formatString = "yyyy/MM/dd hh:mm:ss";

// get a current date object
Date date = Calendar.getInstance().getTime();

// create the formatter
SimpleDateFormat simpleDateFormat = new SimpleDateFormat(formatString);

// format the date
String formattedDate = simpleDateFormat.format(date);

// print it
System.out.println(formattedDate);

// single-line version of all above code
System.out.println(new SimpleDateFormat("yyyy/MM/dd
hh:mm:ss").format(Calendar.getInstance().getTime()));
```

のきを**Date**オブジェクトにする

このメソッドをすると、のされたを `Date` オブジェクトにできます。

```
/**
 * Parses the date using the given format.
 *
 * @param formattedDate the formatted date string
 * @param dateFormat the date format which was used to create the string.
 * @return the date
 */
public static Date parseDate(String formattedDate, String dateFormat) {
    Date date = null;
    SimpleDateFormat objDf = new SimpleDateFormat(dateFormat);
    try {
        date = objDf.parse(formattedDate);
    } catch (ParseException e) {
        // Do what ever needs to be done with exception.
    }
}
```

```

    }
    return date;
}

```

のの

JavaのDateクラスにはいくつかのコンストラクタがありますが、ほとんどのものはです。Dateインスタンスをするためののけれなは、のコンストラクタをするか、longベースからミリをすことです。のをしているや、のDateインスタンスをにっているをき、どちらもです。

しいをするには、Calendarインスタンスがです。そこから、Calendarインスタンスをなにできます。

```
Calendar c = Calendar.getInstance();
```

のにされたしいCalendarインスタンスをします。カレンダーには、とをするや、にするがあります。このは、のにします。

```

c.set(1974, 6, 2, 8, 0, 0);
Date d = c.getTime();

```

getTimeメソッドは、なDateインスタンスをします。カレンダーセットメソッドは1つのフィールドのみをし、それらはすべてしないことにしてください。つまり、をすると、のフィールドはされません。

とし

くの、このコードスニペットはをたしますが、/の2つのなはされていないことにしてください。

- (1974, 6, 2, 8, 0, 0) 1974,6,2,8,0,0 (1974, 6, 2, 8, 0, 0)パラメータは、デフォルトのタイムゾーンでされ、のでされ、
- ミリはゼロにされず、カレンダーインスタンスのにシステムクロックからされます。

Java 8のLocalDateオブジェクトとLocalDateTimeオブジェクト

Dateオブジェクトはのとのをすため、DateオブジェクトとLocalDateオブジェクトはにはできません。LocalDateオブジェクトにはまたはタイムゾーンがまれていません。ただし、ではなくののみをにするは、2つのですとです。

LocalDateをします。

```

// Create a default date
LocalDate lDate = LocalDate.now();

// Creates a date from values
lDate = LocalDate.of(2017, 12, 15);

// create a date from string

```



```
lDate = LocalDate.parse("2017-12-15");

// creates a date from zone
LocalDate.now(ZoneId.systemDefault());
```

LocalDateTimeをします。

```
// Create a default date time
LocalDateTime lDateTime = LocalDateTime.now();

// Creates a date time from values
lDateTime = LocalDateTime.of(2017, 12, 15, 11, 30);

// create a date time from string
lDateTime = LocalDateTime.parse("2017-12-05T11:30:30");

// create a date time from zone
LocalDateTime.now(ZoneId.systemDefault());
```

LocalDate to Dateおよびその

```
Date date = Date.from(Instant.now());
ZoneId defaultZoneId = ZoneId.systemDefault();

// Date to LocalDate
LocalDate localDate = date.toInstant().atZone(defaultZoneId).toLocalDate();

// LocalDate to Date
Date.from(localDate.atStartOfDay(defaultZoneId).toInstant());
```

LocalDateTimeを**Date**に、またはそのにします。

```
Date date = Date.from(Instant.now());
ZoneId defaultZoneId = ZoneId.systemDefault();

// Date to LocalDateTime
LocalDateTime localDateTime = date.toInstant().atZone(defaultZoneId).toLocalDateTime();

// LocalDateTime to Date
Date out = Date.from(localDateTime.atZone(defaultZoneId).toInstant());
```

タイムゾーンとjava.util.Date

java.util.Dateオブジェクトにはタイムゾーンがありません。

- のタイムゾーンをするはありません
- Dateオブジェクトのタイムゾーンをするはありません
- new Date() デフォルトコンストラクタでされたDateオブジェクトは、システムのデフォルトタイムゾーンなのでされます

ただし、java.text.SimpleDateFormatをすると、Dateオブジェクトでされたのでされるをのタイムゾーンにすることができます。

```

Date date = new Date();
//print default time zone
System.out.println(TimeZone.getDefault().getDisplayName());
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss"); //note: time zone not in
format!
//print date in the original time zone
System.out.println(sdf.format(date));
//current time in London
sdf.setTimeZone(TimeZone.getTimeZone("Europe/London"));
System.out.println(sdf.format(date));

```

```

Central European Time
2016-07-21 22:50:56
2016-07-21 21:50:56

```

java.util.Dateをjava.sql.Dateにする

、 java.util.Date から java.sql.Date へのは、Dateオブジェクトをデータベースにきむがあるにです。

java.sql.Date はミリのラッパーで、 JDBCによって SQL DATE をするためにされ JDBC

のでは、 java.util.Date() コンストラクタをして、Dateオブジェクトをし、それをしていミリまでのをします。このは、 java.sql.Date オブジェクトをす convert(java.util.Date utilDate) メソッドでされます。

```

public class UtilToSqlConversion {

    public static void main(String args[])
    {
        java.util.Date utilDate = new java.util.Date();
        System.out.println("java.util.Date is : " + utilDate);
        java.sql.Date sqlDate = convert(utilDate);
        System.out.println("java.sql.Date is : " + sqlDate);
        DateFormat df = new SimpleDateFormat("dd/MM/YYYY - hh:mm:ss");
        System.out.println("dateFormatted date is : " + df.format(utilDate));
    }

    private static java.sql.Date convert(java.util.Date uDate) {
        java.sql.Date sDate = new java.sql.Date(uDate.getTime());
        return sDate;
    }

}

```

```

java.util.Date is : Fri Jul 22 14:40:35 IST 2016
java.sql.Date is : 2016-07-22
dateFormatted date is : 22/07/2016 - 02:40:35

```

java.util.Date はとものがあり java.sql.Date が、 java.sql.Date はしかありません

Dateのだけをするには、LocalTimeをします。いくつかのでLocalTimeオブジェクトをインスタンスできます

```
1. LocalTime time = LocalTime.now();
2. time = LocalTime.MIDNIGHT;
3. time = LocalTime.NOON;
4. time = LocalTime.of(12, 12, 45);
```

`LocalTime`は`toString`メソッドがみまれているので、フォーマットをにうまくできます。

```
System.out.println(time);
```

`LocalTime`オブジェクトから、、、およびナノを、、、することもできます。

```
time.plusMinutes(1);
time.getMinutes();
time.minusMinutes(1);
```

のコードをして`Date`オブジェクトにできます。

```
LocalTime lTime = LocalTime.now();
Instant instant = lTime.atDate(LocalDate.of(A_YEAR, A_MONTH, A_DAY)).
    atZone(ZoneId.systemDefault()).toInstant();
Date time = Date.from(instant);
```

このクラスはアラームクラスをシミュレートするためのタイマークラスでにうまくします。

オンラインでクラスをむ <https://riptutorial.com/ja/java/topic/164/クラス>

169: と java.time。 *

Examples

な

のをします。

```
LocalDate.now()
```

のをします。

```
LocalDate y = LocalDate.now().minusDays(1);
```

のをする

```
LocalDate t = LocalDate.now().plusDays(1);
```

のをします。

```
LocalDate t = LocalDate.of(1974, 6, 2, 8, 30, 0, 0);
```

plus と minus メソッドに plus で、 `LocalDate` インスタンスのフィールドをするためにできるの "with" メソッドがあります。

```
LocalDate.now().withMonth(6);
```

のでは、が6にされたしいインスタンスがされますこれは、 `setMonth` が65に0にインデックスけされた `java.util.Date` とはなります。

`LocalDate`のはの`LocalDate`インスタンスをすため、これらのメソッドはしているもあります。

```
LocalDate ld = LocalDate.now().plusDays(1).plusYears(1);
```

これは、から1ののになります。

タイムゾーンのない

```
LocalDateTime dateTime = LocalDateTime.of(2016, Month.JULY, 27, 8, 0);
LocalDateTime now = LocalDateTime.now();
LocalDateTime parsed = LocalDateTime.parse("2016-07-27T07:00:00");
```

タイムゾーンをむと

```

ZoneId zoneId = ZoneId.of("UTC+2");
ZonedDateTime dateTime = ZonedDateTime.of(2016, Month.JULY, 27, 7, 0, 0, 235, zoneId);
ZonedDateTime composition = ZonedDateTime.of(LocalDate.now(), LocalTime.now(), zoneId);
ZonedDateTime now = ZonedDateTime.now(); // Default time zone
ZonedDateTime parsed = ZonedDateTime.parse("2016-07-27T07:00:00+01:00[Europe/Stockholm]");

```

オフセットをむとつまり、DSTのはされません

```

ZoneOffset zoneOffset = ZoneOffset.ofHours(2);
OffsetDateTime dateTime = OffsetDateTime.of(2016, 7, 27, 7, 0, 0, 235, zoneOffset);
OffsetDateTime composition = OffsetDateTime.of(LocalDate.now(), LocalTime.now(), zoneOffset);
OffsetDateTime now = OffsetDateTime.now(); // Offset taken from the default ZoneId
OffsetDateTime parsed = OffsetDateTime.parse("2016-07-27T07:00:00+02:00");

```

の

```

LocalDate tomorrow = LocalDate.now().plusDays(1);
LocalDateTime anHourFromNow = LocalDateTime.now().plusHours(1);
Long daysBetween = java.time.temporal.ChronoUnit.DAYS.between(LocalDate.now(),
LocalDate.now().plusDays(3)); // 3
Duration duration = Duration.between(Instant.now(), ZonedDateTime.parse("2016-07-
27T07:00:00+01:00[Europe/Stockholm]"))

```

インスタント

をします。 Unixのタイムスタンプをむラッパーとえることができます。

```

Instant now = Instant.now();
Instant epoch1 = Instant.ofEpochMilli(0);
Instant epoch2 = Instant.parse("1970-01-01T00:00:00Z");
java.time.temporal.ChronoUnit.MICROS.between(epoch1, epoch2); // 0

```

Date Time APIのさまざまなクラスの

のも、そののをするためになをっています。

```

import java.time.Clock;
import java.time.Duration;
import java.time.Instant;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.util.TimeZone;

public class SomeMethodsExamples {

    /**
     * Has the methods of the class {@link LocalDateTime}
     */
    public static void checkLocalDateTime() {
        LocalDateTime localDateTime = LocalDateTime.now();
        System.out.println("Local Date time using static now() method ::: >>> "

```

```

        + localDateTime);

LocalDateTime ldt1 = LocalDateTime.now(ZoneId.of(ZoneId.SHORT_IDS
        .get("AET")));
System.out
        .println("LOCAL TIME USING now(ZoneId zoneId) method ::: >>>>"
                + ldt1);

LocalDateTime ldt2 = LocalDateTime.now(Clock.system(ZoneId
        .of(ZoneId.SHORT_IDS.get("PST"))));
System.out
        .println("Local TIME USING now(Clock.system(ZoneId.of())) ::: >>>> "
                + ldt2);

System.out
        .println("Following is a static map in ZoneId class which has mapping of short
timezone names to their Actual timezone names");
System.out.println(ZoneId.SHORT_IDS);
}

/**
 * This has the methods of the class {@link LocalDate}
 */
public static void checkLocalDate() {
    LocalDate localDate = LocalDate.now();
    System.out.println("Gives date without Time using now() method. >> "
            + localDate);
    LocalDate localDate2 = LocalDate.now(ZoneId.of(ZoneId.SHORT_IDS
            .get("ECT")));
    System.out
            .println("now() is overridden to take ZoneID as parametere using this we can get
the same date under different timezones. >> "
                    + localDate2);
}

/**
 * This has the methods of abstract class {@link Clock}. Clock can be used
 * for time which has time with {@link TimeZone}.
 */
public static void checkClock() {
    Clock clock = Clock.systemUTC();
    // Represents time according to ISO 8601
    System.out.println("Time using Clock class : " + clock.instant());
}

/**
 * This has the {@link Instant} class methods.
 */
public static void checkInstant() {
    Instant instant = Instant.now();

    System.out.println("Instant using now() method :: " + instant);

    Instant ins1 = Instant.now(Clock.systemUTC());

    System.out.println("Instants using now(Clock clock) :: " + ins1);
}

/**

```

```

* This class checks the methods of the {@link Duration} class.
*/
public static void checkDuration() {
    // toString() converts the duration to PTnHnMnS format according to ISO
    // 8601 standard. If a field is zero its ignored.

    // P is the duration designator (historically called "period") placed at
    // the start of the duration representation.
    // Y is the year designator that follows the value for the number of
    // years.
    // M is the month designator that follows the value for the number of
    // months.
    // W is the week designator that follows the value for the number of
    // weeks.
    // D is the day designator that follows the value for the number of
    // days.
    // T is the time designator that precedes the time components of the
    // representation.
    // H is the hour designator that follows the value for the number of
    // hours.
    // M is the minute designator that follows the value for the number of
    // minutes.
    // S is the second designator that follows the value for the number of
    // seconds.

    System.out.println(Duration.ofDays(2));
}

/**
 * Shows Local time without date. It doesn't store or represent a date and
 * time. Instead its a representation of Time like clock on the wall.
 */
public static void checkLocalTime() {
    LocalDateTime localTime = LocalDateTime.now();
    System.out.println("LocalTime :: " + localTime);
}

/**
 * A date time with Time zone details in ISO-8601 standards.
 */
public static void checkZonedDateTime() {
    ZonedDateTime zonedDateTime = ZonedDateTime.now(ZoneId
        .of(ZoneId.SHORT_IDS.get("CST")));
    System.out.println(zonedDateTime);
}
}

```

の

Java 8よりでは、`java.text`パッケージに`DateFormat`クラスと`SimpleDateFormat`クラスがありました。このレガシーコードはいつかされけます。

しかし、Java 8は、とをうなアプローチをしています。

とでは、に`String`オブジェクトを`DateTimeFormatter`し、にそれをまたはにします。

```

import java.time.*;
import java.time.format.*;

class DateTimeFormat
{
    public static void main(String[] args) {

        //Parsing
        String pattern = "d-MM-yyyy HH:mm";
        DateTimeFormatter dtF1 = DateTimeFormatter.ofPattern(pattern);

        LocalDateTime ldp1 = LocalDateTime.parse("2014-03-25T01:30"), //Default format
            ldp2 = LocalDateTime.parse("15-05-2016 13:55",dtF1); //Custom format

        System.out.println(ldp1 + "\n" + ldp2); //Will be printed in Default format

        //Formatting
        DateTimeFormatter dtF2 = DateTimeFormatter.ofPattern("EEE d, MMMM, yyyy HH:mm");

        DateTimeFormatter dtF3 = DateTimeFormatter.ISO_LOCAL_DATE_TIME;

        LocalDateTime ldtf1 = LocalDateTime.now();

        System.out.println(ldtf1.format(dtF2) + "\n"+ldtf1.format(dtF3));
    }
}

```

なとして、カスタムパターンをするわりに、あらかじめされたフォーマッタをすることをおめします。あなたのコードはよりにえ、ISO8061のはいなくにあなたをけます。

2つのLocalDateのをする

LocalDateとChronoUnitする

```

LocalDate d1 = LocalDate.of(2017, 5, 1);
LocalDate d2 = LocalDate.of(2017, 5, 18);

```

は、ChronoUnitChronoUnit betweenのメソッドが2つのTemporalをパラメータとしてChronoUnitので、なくすことができるので、LocalDateインスタンス

```

long days = ChronoUnit.DAYS.between(d1, d2);
System.out.println( days );

```

オンラインでとjava.time。*をむ <https://riptutorial.com/ja/java/topic/4813/>と-java-time---

170: インタフェース

き

Java 8では、インタフェースは、Objectのメソッドにも、メソッドを1つしかたないインタフェースです。 [JLS§9.8](#)をしてください。 [インタフェース](#)。

Examples

によるJava Runtime Libraryのインタフェースのリスト

パラメータタイプ	りの	インタフェース
ランナブル		
	T	サプライヤー
	ブール	ブールサプライヤ
	int	IntSupplier
	いです	LongSupplier
	ダブル	DoubleSupplier
T		コンシューマー<T>
T	T	UnaryOperator <T>
T	R	<T、 R>
T	ブール	<T>
T	int	ToIntFunction <T>
T	いです	ToLongFunction <T>
T	ダブル	ToDoubleFunction <T>
T、 T	T	BinaryOperator <T>
T、 U		BiConsumer <T、 U>
T、 U	R	BiFunction <T、 U、 R>
T、 U	ブール	BiPredicate <T、 U>

パラメータタイプ	りの	インタフェース
T、U	int	ToIntBiFunction <T、 U>
T、U	いです	ToLongBiFunction <T、 U>
T、U	ダブル	ToDoubleBiFunction <T、 U>
T、int		ObjIntConsumer <T>
T、long		ObjLongConsumer <T>
T、ダブル		ObjDoubleConsumer <T>
int		IntConsumer
int	R	IntFunction <R>
int	ブール	IntPredicate
int	int	IntUnaryOperator
int	いです	IntToLongFunction
int	ダブル	IntToDoubleFunction
int、int	int	IntBinaryOperator
いです		ロングコンシューマー
いです	R	LongFunction <R>
いです	ブール	LongPredicate
いです	int	LongToIntFunction
いです	いです	LongUnaryOperator
いです	ダブル	LongToDoubleFunction
い、い	いです	LongBinaryOperator
ダブル		DoubleConsumer
ダブル	R	DoubleFunction <R>
ダブル	ブール	
ダブル	int	DoubleToIntFunction

パラメータタイプ	りの	インタフェース
ダブル	いです	DoubleToLongFunction
ダブル	ダブル	DoubleUnaryOperator
ダブル、ダブル	ダブル	DoubleBinaryOperator

オンラインでインタフェースをむ <https://riptutorial.com/ja/java/topic/10001/インタフェース>

171:

き

は、パターンにされたなをして、のやをまたはするのにつなシーケンスです。Javaは、`java.util.regex`パッケージをしたのをサポートしています。このトピックでは、をJavaでどのようにするがあるかのについて、がくのことをし、するのをけます。

- パターン `patternName = Pattern.compile;`
- マッチャー `matcherName = patternName.matcher(textToSearch);`
- `matcherName.matches()` `textToSearch`がとにするはtrueをします
- `matcherName.find()`とするののインスタンスを`textToSearch`でします。のびしは、のりのをします。
- `matcherName.group(groupNum)`キャプチャグループのをします
- `matcherName.group(groupName)`きキャプチャグループJava 7+のをします。

Regexをするに、のインポートをするがあります

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
```

とし

Javaでは、バックスラッシュはバックスラッシュでエスケープされるため、のバックスラッシュはバックスラッシュとしてするがあります。1つのバックスラッシュとをさせるために、2つのバックスラッシュをエスケープするがあるは、それを4つのバックスラッシュとしてするがあります。

なシンボルの

キャラクター	
*	のまたはを0マッチさせる
+	のまたはを1マッチさせる
?	のまたはを0または1マッチさせる

トピックには、 にするがまれています。

Examples

グループの

からのをすることがあるは、のグループをできます。

このでは、なからめます。

```
\d{3}-\d{3}-\d{4}
```

カッコがにされている、カッコのセットはキャプチャグループとみなされます。この、きキャプチャグループとばれるものをしてしています。

```
(\d{3})-(\d{3})-(\d{4})
^-----^ ^-----^ ^-----^
Group 1 Group 2 Group 3
```

Javaでするに、のにつて、バックスラッシュをエスケープしてのパターンにすることをれてはなりません。

```
"(\\d{3})-(\\d{3})-(\\d{4})"
```

たちは、にるためにパターンをコンパイルするがありPatternし、々はなMatcherパターンとのにマッチします

```
Pattern phonePattern = Pattern.compile("(\\d{3})-(\\d{3})-(\\d{4})");
Matcher phoneMatcher = phonePattern.matcher("abcd800-555-1234wxyz");
```

に、Matcherはとするのをつけるがあります

```
phoneMatcher.find();
```

さて、groupメソッドをとって、からデータをすることが出来ます

```
String number = phoneMatcher.group(0); //"800-555-1234" (Group 0 is everything the regex
matched)
String aCode = phoneMatcher.group(1); //"800"
String threeDigit = phoneMatcher.group(2); //"555"
String fourDigit = phoneMatcher.group(3); //"1234"
```

Matcher.group() Matcher.group(0) わりにMatcher.group(0) をできます。

Java SE 7

Java 7では、きキャプチャグループがされました。きキャプチャグループは、のいたキャプチャ

グループと同じようにしますがではなくがいています、わずかながあります。きキャプチャグループをすると、みやすさがします。

のコードをきグループをするようにすることができます

```
(?<AreaCode>\d{3})-(\d{3})-(\d{4})
^-----^ ^-----^ ^-----^
AreaCode      Group 2 Group 3
```

"AreaCode"のをするには、わりにのコードをします。

```
String aCode = phoneMatcher.group("AreaCode"); //"800"
```

パターンをフラグでコンパイルしてカスタムをする **regex** の

`Pattern` はフラグでコンパイルできますが、が `String` としてされているは、インラインをしてください

```
Pattern pattern = Pattern.compile("foo.", Pattern.CASE_INSENSITIVE | Pattern.DOTALL);
pattern.matcher("FOO\n").matches(); // Is true.

/* Had the regex not been compiled case insensitively and singlelined,
 * it would fail because FOO does not match /foo/ and \n (newline)
 * does not match /. /.
 */

Pattern anotherPattern = Pattern.compile("(?si)foo");
anotherPattern.matcher("FOO\n").matches(); // Is true.

"foOt".replaceAll("(?si)foo", "ca"); // Returns "cat".
```

エスケープ

に

にの `?+|` などをするには、それらをエスケープするがあります。なでは、これはバックスラッシュによってわれます、。ただし、Javaではながあるため、`\\` をするがあります。

これらの2つのはうまくいかないでしょう

```
"???.replaceAll ("?", "!"); //java.util.regex.PatternSyntaxException
"???.replaceAll ("\\?", "!"); //Invalid escape sequence
```

このはします

```
"???.replaceAll ("\\?", "!"); //"!!!"
```

パイプりの

これはされるをしません

```
"a|b".split ("|"); // [a, |, b]
```

されるがされます。

```
"a|b".split ("\\|"); // [a, b]
```

バックスラッシュをエスケープする、

これによりエラーがします

```
"\\".matches("\\"); // PatternSyntaxException  
"\\".matches("\\\\"); // Syntax Error
```

これはします

```
"\\".matches("\\\\\\\\"); // true
```

のリテラルとのマッチング

のであるをさせるがあるは、パターンのすべてまたはをリテラルとしてマークすることができます。

`\Q`はのリテラルのをします。 `\E`はのリテラルのわりをします。

```
// the following throws a PatternSyntaxException because of the un-closed bracket  
"[123".matches("[123");  
  
// wrapping the bracket in \Q and \E allows the pattern to match as you would expect.  
"[123".matches("\Q[\E123"); // returns true
```

`\Q`と`\E`エスケープシーケンスをえることなくにできるは、 `Pattern.quote()` をすること
`Pattern.quote()`

```
"[123".matches(Pattern.quote("[") + "123"); // returns true
```

されたにしません

されたをまないものにマッチさせるには、なみをうことができます

の `(?!string-to-not-match)`

```
//not matching "popcorn"  
String regexString = "^(?!popcorn).*$";  
System.out.println("[popcorn] " + ("popcorn".matches(regexString) ? "matched!" : "nope!"));  
System.out.println("[unicorn] " + ("unicorn".matches(regexString) ? "matched!" : "nope!"));
```

```
[popcorn] nope!  
[unicorn] matched!
```

バックスラッシュのマッチング

でバックスラッシュをマッチさせたいは、それをエスケープする必要があります。

バックスラッシュはのエスケープです。 \\をして、ののバックスラッシュをすることができます。

しかし、バックスラッシュはJavaリテラルのエスケープでもあります。リテラルからをするには、それぞれのバックスラッシュをエスケープする必要があります。リテラルでは、`\\`をして`\`でできます。このは`\`とします。

たとえば、"C:\dir\myfile.txt"のようなをさせることをしてください。([A-Za-z]):\\(.*)がし、ドライブをキャプチャグループとしてします。のバックスラッシュにしてください。

そのパターンをJavaリテラルにするには、のバックスラッシュのそれぞれをエスケープする必要があります。

```
String path = "C:\\dir\\myfile.txt";
System.out.println( "Local path: " + path ); // "C:\dir\myfile.txt"

String regex = "([A-Za-z]):\\\\".*"; // Four to match one
System.out.println("Regex:      " + regex ); // "([A-Za-z]):\\(\\.*)"

Pattern pattern = Pattern.compile( regex );
Matcher matcher = pattern.matcher( path );
if ( matcher.matches() ) {
    System.out.println( "This path is on drive " + matcher.group( 1 ) + ":\n");
    // This path is on drive C:.
}
```

2つのバックスラッシュをマッチさせたいは、リテラルで8をして、で4を2にさせることができます。

```
String path = "\\myhost\\share\\myfile.txt";
System.out.println( "UNC path: " + path ); // \\myhost\share\myfile.txt

String regex = "\\(\\|\\\\)(.+?)\\\\(\\|\\\\)(.*)"; // Eight to match two
System.out.println("Regex:      " + regex ); // \\(\\|\\\\)(.+?)\\\\(\\|\\\\)(.*)

Pattern pattern = Pattern.compile( regex );
Matcher matcher = pattern.matcher( path );

if ( matcher.matches() ) {
    System.out.println( "This path is on host '" + matcher.group( 1 ) + "'.");
    // This path is on host 'myhost'.
}
```

オンラインでをむ <https://riptutorial.com/ja/java/topic/135/>

172: なコンパレータ

- パブリッククラスMyClassは、Comparable <MyClass >
- パブリッククラスMyComparatorは、Comparator <SomeOtherClass >
- public int compareToMyClassその
- public int compareSomeOtherClass o1、SomeOtherClass o2

doubleにするcompareTo(..)メソッドをするは、のはわないでください。

```
public int comareTo(MyClass other) {  
    return (int)(doubleField - other.doubleField); //THIS IS BAD  
}
```

(int)キャストによるりてにより、メソッドがまたはのではなく、0をとってすことがあり、その、およびソートのバグにつながるがあります。

わりに、Double.compareをのようにするのがもなしいです。

```
public int comareTo(MyClass other) {  
    return Double.compare(doubleField, other.doubleField); //THIS IS GOOD  
}
```

Comparable<T>ジェネリックでないバージョン、にComparable [はJava 1.2しています](#)。レガシーコードとのインタフェースにも、ジェネリックのComparable<T>、にキャストするがないため、するがにれています。

クラスがそれにすることは、のようになです。

```
public class A implements Comparable<A>
```

このパラダイムかられるはありますが、そうするときにはにしてください。

Comparator<T>は、そのクラスがComparable<T>しているでも、そのクラスのインスタンスでできます。この、Comparatorのロジックがされます。Comparableでされただけはされます。

Examples

Comparableをってリストをソートするまたはコンパレータ

たちはPersonをですクラスをっているとしましょう。これをうためのクラスをし、なequals hashCodeとhashCodeメソッドをしました。

```

public class Person {

    private final String lastName; //invariant - nonnull
    private final String firstName; //invariant - nonnull

    public Person(String firstName, String lastName){
        this.firstName = firstName != null ? firstName : "";
        this.lastName = lastName != null ? lastName : "";
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public String toString() {
        return lastName + ", " + firstName;
    }

    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Person)) return false;
        Person p = (Person)o;
        return firstName.equals(p.firstName) && lastName.equals(p.lastName);
    }

    @Override
    public int hashCode() {
        return Objects.hash(firstName, lastName);
    }
}

```

のシナリオのように、`Person` オブジェクトのリストをでソートするとします。

```

public static void main(String[] args) {
    List<Person> people = Arrays.asList(new Person("John", "Doe"),
                                         new Person("Bob", "Dole"),
                                         new Person("Ronald", "McDonald"),
                                         new Person("Alice", "McDonald"),
                                         new Person("Jill", "Doe"));

    Collections.sort(people); //This currently won't work.
}

```

ながら、マークされているように、はコンパイルされません。 `Collections.sort(..)` は、リストの
がしている、またはのカスタムメソッドがされているにのみ、リストのソートを行っています。

あなたがのリストをべえるようにめられたなら `1,3,5,4,2`、えは `1,2,3,4,5` とってもありません。
これは、Java とにはともがけをつためです。これは、のデフォルトベースです。 `Person` クラスに
けをえるために、 `compareTo(Person p):` メソッドのが `Comparable<Person>` をし `compareTo(Person p):`

```

public class Person implements Comparable<Person> {

    private final String lastName; //invariant - nonnull
    private final String firstName; //invariant - nonnull

```

```

public Person(String firstName, String lastName) {
    this.firstName = firstName != null ? firstName : "";
    this.lastName = lastName != null ? lastName : "";
}

public String getFirstName() {
    return firstName;
}

public String getLastName() {
    return lastName;
}

public String toString() {
    return lastName + ", " + firstName;
}

@Override
public boolean equals(Object o) {
    if (! (o instanceof Person)) return false;
    Person p = (Person)o;
    return firstName.equals(p.firstName) && lastName.equals(p.lastName);
}

@Override
public int hashCode() {
    return Objects.hash(firstName, lastName);
}

@Override
public int compareTo(Person other) {
    // If this' lastName and other's lastName are not comparably equivalent,
    // Compare this to other by comparing their last names.
    // Otherwise, compare this to other by comparing their first names
    int lastNameCompare = lastName.compareTo(other.lastName);
    if (lastNameCompare != 0) {
        return lastNameCompare;
    } else {
        return firstName.compareTo(other.firstName);
    }
}
}

```

さて、えられたなメソッドはしくします

```

public static void main(String[] args) {
    List<Person> people = Arrays.asList(new Person("John", "Doe"),
                                         new Person("Bob", "Dole"),
                                         new Person("Ronald", "McDonald"),
                                         new Person("Alice", "McDonald"),
                                         new Person("Jill", "Doe"));

    Collections.sort(people); //Now functions correctly

    //people is now sorted by last name, then first name:
    // --> Jill Doe, John Doe, Bob Dole, Alice McDonald, Ronald McDonald
}

```

ただし、クラス `Person` をしたくないやできないは、の2つの `Person` オブジェクトのをするカスタム

Comparator<T>をすることができます。 circle, square, rectangle, triangle, hexagon リストをソートするようされた、コーナーのについてそのリストをソートするようにめられたは、そうすることができます。ちょうど、をすることは、Javaに、2つのできないオブジェクトをするをします。

```
public class PersonComparator implements Comparator<Person> {

    public int compare(Person p1, Person p2) {
        // If p1's lastName and p2's lastName are not comparably equivalent,
        // Compare p1 to p2 by comparing their last names.
        // Otherwise, compare p1 to p2 by comparing their first names
        if (p1.getLastName().compareTo(p2.getLastName()) != 0) {
            return p1.getLastName().compareTo(p2.getLastName());
        } else {
            return p1.getFirstName().compareTo(p2.getFirstName());
        }
    }
}

//Assume the first version of Person (that does not implement Comparable) is used here
public static void main(String[] args) {
    List<Person> people = Arrays.asList(new Person("John", "Doe"),
                                        new Person("Bob", "Dole"),
                                        new Person("Ronald", "McDonald"),
                                        new Person("Alice", "McDonald"),
                                        new Person("Jill", "Doe"));

    Collections.sort(people); //Illegal, Person doesn't implement Comparable.
    Collections.sort(people, new PersonComparator()); //Legal

    //people is now sorted by last name, then first name:
    // --> Jill Doe, John Doe, Bob Dole, Alice McDonald, Ronald McDonald
}
```

Comparatorは、のクラスとして/することもできます

```
//Assume the first version of Person (that does not implement Comparable) is used here
public static void main(String[] args) {
    List<Person> people = Arrays.asList(new Person("John", "Doe"),
                                        new Person("Bob", "Dole"),
                                        new Person("Ronald", "McDonald"),
                                        new Person("Alice", "McDonald"),
                                        new Person("Jill", "Doe"));

    Collections.sort(people); //Illegal, Person doesn't implement Comparable.

    Collections.sort(people, new PersonComparator()); //Legal

    //people is now sorted by last name, then first name:
    // --> Jill Doe, John Doe, Bob Dole, Alice McDonald, Ronald McDonald

    //Anonymous Class
    Collections.sort(people, new Comparator<Person>() { //Legal
        public int compare(Person p1, Person p2) {
            //Method code...
        }
    });
}
```

ラムダコンパレータ

Java 8のでは、コンパレータはラムダ

```
//Lambda
Collections.sort(people, (p1, p2) -> { //Legal
    //Method code....
});
```

コンパレータのデフォルトメソッド

さらに、**Comparator**をするための**Comparator**インターフェイスには、いデフォルトメソッドがあります。は、`lastName`と`firstName`をし`firstName`。

```
Collections.sort(people, Comparator.comparing(Person::getLastName)
    .thenComparing(Person::getFirstName));
```

コンパレータのをする

のものにしてにすることができる`reversedMethod`ににされます。

compareTo メソッドと**compare** メソッド

Comparable<T>インターフェイスには、1つのメソッドがです。

```
public interface Comparable<T> {

    public int compareTo(T other);

}
```

また、**Comparator<T>**インターフェイスには、1つのメソッドがです。

```
public interface Comparator<T> {

    public int compare(T t1, T t2);

}
```

これらの2つののは、1つのマイナーいが、にじことをう`compareTo`し`this`する`other`のにし、`compareTo`する`t2`ではなく、いやりのすべてについて`this`。

そのいとはに、2つのにものがあります。に`compareTo`の、このオブジェクトをされたオブジェクトとしてします。このオブジェクトがされたオブジェクトよりもさい、しい、またはきいは、

の、ゼロまたはのをします。したがって、 a と b の a

- $a < b$ 、`a.compareTo(b)` および `compare(a,b)` がのをし、`b.compareTo(a)` および `compare(b,a)` がのをすがある
- $a > b$ 、`a.compareTo(b)` および `compare(a,b)` がのをし、`b.compareTo(a)` および `compare(b,a)` がのをすがある
- しい a b のために、すべてのはすがあります₀。

とソート

2つの`Collections.sort()` メソッドがあります

- かかる`List<T>` パラメータとして T し、きしななければならない`compareTo()` ソートをする。
- として`List`と`Comparator`をとり、`Comparator`がソートをするものです。

まず、`Comparable`をする`Person`クラスをにします。

```
public class Person implements Comparable<Person> {
    private String name;
    private int age;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public int compareTo(Person o) {
        return this.getAge() - o.getAge();
    }
    @Override
    public String toString() {
        return this.getAge()+"-"+this.getName();
    }
}
```

のクラスをして、`compareTo()` メソッドのオーバーライドによってされたのけでリストをソートするをにします。

```
//-- usage
List<Person> pList = new ArrayList<Person>();
Person p = new Person();
p.setName("A");
p.setAge(10);
pList.add(p);
```

```

p = new Person();
p.setName("Z");
p.setAge(20);
pList.add(p);
p = new Person();
p.setName("D");
p.setAge(30);
pList.add(p);

//-- natural sorting i.e comes with object implementation, by age
Collections.sort(pList);

System.out.println(pList);

```

のインラインコンパレータをして、ComparableをしていないListをソートするか、この、けのでListをソートするのはのとおりです。

```

//-- explicit sorting, define sort on another property here goes with name
Collections.sort(pList, new Comparator<Person>() {

    @Override
    public int compare(Person o1, Person o2) {
        return o1.getName().compareTo(o2.getName());
    }
});
System.out.println(pList);

```

マップエントリのソート

Java 8では、マップのソートをにするためにMap.EntryインターフェースにデフォルトのメソッドがありMap.Entry。

Java SE 8

```

Map<String, Integer> numberOfEmployees = new HashMap<>();
numberOfEmployees.put("executives", 10);
numberOfEmployees.put("human ressources", 32);
numberOfEmployees.put("accounting", 12);
numberOfEmployees.put("IT", 100);

// Output the smallest departement in terms of number of employees
numberOfEmployees.entrySet().stream()
    .sorted(Map.Entry.comparingByValue())
    .limit(1)
    .forEach(System.out::println); // outputs : executives=10

```

もちろん、これらはストリームAPIのでもできます。

Java SE 8

```

List<Map.Entry<String, Integer>> entries = new ArrayList<>(numberOfEmployees.entrySet());
Collections.sort(entries, Map.Entry.comparingByValue());

```

メソッドをしたコンパレータの

```
Comparator.comparing(Person::getName)
```

これは、この`Person`をとする`Person`クラスのコンパレータをします。メソッドバージョンをして`long`、`int`、`double`をすることもできます。え

```
Comparator.comparingInt(Person::getAge)
```

をするコンパレータをするには、`reverse` `reversed()` メソッドをします。

```
Comparator.comparing(Person::getName).reversed()
```

コンパレータのチェーン

```
Comparator.comparing(Person::getLastName).thenComparing(Person::getFirstName)
```

これは、`firs`がとしてファーストネームとするコンパレータをします。くのコンパレータをにしてチェーンすることができます。

オンラインでなコンパレータをむ <https://riptutorial.com/ja/java/topic/3137/>なコンパレータ

173:

き

Javaでは、はJavaソースコードにできるメタデータのです。プログラムにまれていないプログラムにするデータをします。はをけるコードののをえません。クラス、メソッド、、パラメータ、パッケージにをけることができます。

- `@AnnotationName` // 'マーカーアノテーション'パラメータなし
- `@AnnotationNamesomeValue`// 'value'というのパラメータをする
- `@AnnotationNameparam1 = value1`//きパラメータ
- `@AnnotationNameparam1 = value1、 param2 = value2`//のきパラメータ
- `@AnnotationNameparam1 = {1、 2、 3}`//きパラメータ
- `@AnnotationName{value1}`//が 'value'のパラメータとしてをつ

パラメータタイプ

パラメータには、ののと、これらののだけがされます。

- String
- Class
- プリミティブ
-
- タイプ

Examples

ビルト イン

JavaのStandard Editionには、いくつかのアノテーションがあらかじめされています。ですではなく、すぐにできます。コンパイラはメソッド、クラス、コードのなチェックをにします。

@オーバーライド

このはメソッドにされ、このメソッドはスーパークラスのメソッドをオーバーライドするか、スーパークラスのメソッドをするがあるといいます。このアノテーションがのメソッドでされている、コンパイラはエラーをスローします。

コンクリートスーパークラス

```
public class Vehicle {  
    public void drive() {  
        System.out.println("I am driving");  
    }  
}
```

```

    }
}

class Car extends Vehicle {
    // Fine
    @Override
    public void drive() {
        System.out.println("Brrrm, brrm");
    }
}

```

クラス

```

abstract class Animal {
    public abstract void makeNoise();
}

class Dog extends Animal {
    // Fine
    @Override
    public void makeNoise() {
        System.out.println("Woof");
    }
}

```

しません

```

class Logger1 {
    public void log(String logString) {
        System.out.println(logString);
    }
}

class Logger2 {
    // This will throw compile-time error. Logger2 is not a subclass of Logger1.
    // log method is not overriding anything
    @Override
    public void log(String logString) {
        System.out.println("Log 2" + logString);
    }
}

```

なは、メソッドをオーバーライドしているとわれるミスタイプをキャッチすることですが、にはしいメソッドをしています。

```

class Vehicle {
    public void drive() {
        System.out.println("I am driving");
    }
}

class Car extends Vehicle {
    // Compiler error. "dirve" is not the correct method name to override.
    @Override
    public void dirve() {
        System.out.println("Brrrm, brrm");
    }
}

```

```
}
```

`@Override`のはとともにしていることにしてください。

- Java 5では、きメソッドはスーパークラスチェーンでされたメソッドをオーバーライドするがありました。
- Java 6では、きメソッドがクラススーパークラス/インタフェースでされたメソッドをしているにも、しています。

これはコードをJava 5にバックポーティングするときをきこすことがあります。

@Deprecated

これはメソッドをとマークします。これにはいくつかのがあります。

- APIにがあり、するのはではありません。
- APIのはエラーにつながるがありますが、
- APIはのAPIによってわれましたが、
- APIはれですが、
- APIはなものであり、のないをけることがあります。
- またはののみわけをむ。

のなは、、APIのドキュメントにされています。

アノテーションをすると、コンパイラはエラーをさせます。IDEは、このメソッドをらかのでされたものとしてするかもしれません

```
class ComplexAlgorithm {
    @Deprecated
    public void oldSlowUnthreadSafeMethod() {
        // stuff here
    }

    public void quickThreadSafeMethod() {
        // client code should use this instead
    }
}
```

@SuppressWarnings

ほとんどすべての、コンパイラがをするとき、もなアクションはをすることです。いくつかのはジェネリックスコードはなコードをしています、これはではないかもしれません。していてできないをして、せぬをよりにできます。

このアノテーションは、クラス、メソッド、またはにできます。のカテゴリをパラメータとしてります。

```
@SuppressWarnings("deprecation")
public class RiddledWithWarnings {
    // several methods calling deprecated code here
}

@SuppressWarnings("finally")
public boolean checkData() {
    // method calling return from within finally block
}
```

しないもされないように、アノテーションのスコープをなりするがよいでしょう。たとえば、アノテーションのスコープを1にします。

```
ComplexAlgorithm algorithm = new ComplexAlgorithm();
@SuppressWarnings("deprecation") algorithm.slowUnthreadSafeMethod();
// we marked this method deprecated in an example above

@SuppressWarnings("unsafe") List<Integer> list = getUntypeSafeList();
// old library returns, non-generic List containing only integers
```

このアノテーションでサポートされているは、コンパイラごとになるがあります。JLSでは、unchecked deprecation ていunchecked と deprecation のみがにされています。されないのはされます。

@SafeVarargs

のため、 void method(T... t) は void method(Object[] t) され、コンパイラはにvarargsのがタイプセーフであることをできないことをします。えは

```
private static <T> void generatesVarargsWarning(T... lists) {
```

がであるがあります。この、 SafeVarargs アノテーションをして SafeVarargs をけてをすることができます。これはあなたのがでないでもをすことは確かです。

@FunctionalInterface

これは、FunctionalInterfaceをマークするためにされるオプションのです。FunctionalInterfaceにしていないのメソッドをつ、コンパイラがエラーをさせます。

```
@FunctionalInterface
public interface ITrade {
    public boolean check(Trade t);
}

@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

リフレクションによるランタイムアノテーションチェック

JavaのReflection APIをすると、プログラマはにクラスフィールド、メソッド、およびアノテーションにしてさまざまなチェックとをできます。ただし、にがすべてされるようにするには、のにように、 `RetentionPolicy`を`RUNTIME`にするがあります。

```
@interface MyDefaultAnnotation {  
  
}  
  
@Retention(RetentionPolicy.RUNTIME)  
@interface MyRuntimeVisibleAnnotation {  
  
}  
  
public class AnnotationAtRuntimeTest {  
  
    @MyDefaultAnnotation  
    static class RuntimeCheck1 {  
    }  
  
    @MyRuntimeVisibleAnnotation  
    static class RuntimeCheck2 {  
    }  
  
    public static void main(String[] args) {  
        Annotation[] annotationsByType = RuntimeCheck1.class.getAnnotations();  
        Annotation[] annotationsByType2 = RuntimeCheck2.class.getAnnotations();  
  
        System.out.println("default retention: " + Arrays.toString(annotationsByType));  
        System.out.println("runtime retention: " + Arrays.toString(annotationsByType2));  
    }  
}
```

タイプの

タイプは`@interface`されます。パラメータは、のインタフェースのメソッドとにされます。

```
@interface MyAnnotation {  
    String param1();  
    boolean param2();  
    int[] param3(); // array parameter  
}
```

デフォルト

```
@interface MyAnnotation {  
    String param1() default "someValue";  
    boolean param2() default true;  
    int[] param3() default {};  
}
```

メタ

メタは、タイプにできるです。なメタアノテーションは、アノテーションタイプのをします。

@ターゲット

@Targetメタアノテーションは、アノテーションをできるタイプをします。

```
@Target (ElementType.METHOD)
@interface MyAnnotation {
    // this annotation can only be applied to methods
}
```

@Target ({ElementType.FIELD, ElementType.TYPE}) をしてのをできます

な

ElementType	ターゲット	ターゲットの
ANNOTATION_TYPE	アノテーションタイプ	<pre>@Retention (RetentionPolicy.RUNTIME) @interface MyAnnotation</pre>
コンストラクタ	コンストラクタ	<pre>@MyAnnotation public MyClass() {}</pre>
フィールド	フィールド、	<pre>@XmlAttribute private int count;</pre>
LOCAL_VARIABLE	メソッドの	<pre>for (@LoopVariable int i = 0; i < 100; i++) { @Unused String resultVariable; }</pre>
パッケージ	<code>package</code> in <code>package-</code> <code>info.java</code>	<pre>@Deprecated package very.old;</pre>
	メソッド	<pre>@XmlElement</pre>

ElementType	ターゲット	ターゲットの
		<pre>public int getCount() {...}</pre>
パラメータ	メソッド/コンストラクタのパラメータ	<pre>public Rectangle(@NamedArg("width") double width, @NamedArg("height") double height) { ... }</pre>
タイプ	クラス、インタフェース、	<pre>@XmlRootElement public class Report {}</pre>

Java SE 8

ElementType	ターゲット	ターゲットの
TYPE_PARAMETER	タイプパラメータ	<pre>public <@MyAnnotation T> void f(T t) {}</pre>
TYPE_USE	タイプの	<pre>Object o = "42"; String s = (@MyAnnotation String) o;</pre>

@

`@Retention` メタアノテーションは、アプリケーションコンパイルプロセスまたはのアノテーションをします。デフォルトでは、アノテーションは `.class` ファイルにまれています。にはされません。にアノテーションにアクセスできるようにするには、そのアノテーションに `RetentionPolicy.RUNTIME` をするがあります。

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation {
    // this annotation can be accessed with reflections at runtime
}
```

な

RetentionPolicy	
クラス	アノテーションは <code>.class</code> ファイルでできますが、にはできません
ランタイム	アノテーションはにで、リフレクションをしてアクセスできます

アノテーションはコンパイルにできますが、`.class` ファイルにはされません。は、え、プロセッサによってすることができる。

@Documented

`@Documented` メタアノテーションは、[javadoc](#)のようなAPIドキュメントジェネレータによってがされるべきアノテーションをマークするためにされます。は、ありません。 `@Documented` をすると、アノテーションをするすべてのクラスがされたドキュメントページにそのアノテーションをリストします。 `@Documented` をしないと、ドキュメンテーションのをするクラスをすることはできません。

@されました

`@Inherited` メタアノテーションは、クラスにされるアノテーションにしています。は、ありません。アノテーションを `@Inherited` としてマークすると、アノテーションクエリがするがされます。

- されていないの、はされるクラスのみをします。
- されたアノテーションについては、アノテーションのインスタスがつかるまでスーパークラスチェーンにもチェックされます。

スーパークラスのみがされることにしてください。クラスのインターフェースにアタッチされたはされます。

@りし

`@Repeatable` メタアノテーションはJava 8でされました。アノテーションののインスタスがアノテーションのターゲットにアタッチできることをします。このメタにははありません。

にをする

[Reflection](#) をして `Annotation` がされているメソッドまたはフィールドまたはクラスをフェッチし、の `プロパティ` をフェッチすることで、`Annotation` のの `プロパティ` をできます。

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation {
    String key() default "foo";
    String value() default "bar";
}

class AnnotationExample {
    // Put the Annotation on the method, but leave the defaults
    @MyAnnotation
    public void testDefaults() throws Exception {
        // Using reflection, get the public method "testDefaults", which is this method with
        no args
    }
}
```



```

Method method = AnnotationExample.class.getMethod("testDefaults", null);

// Fetch the Annotation that is of type MyAnnotation from the Method
MyAnnotation annotation = (MyAnnotation)method.getAnnotation(MyAnnotation.class);

// Print out the settings of the Annotation
print(annotation);
}

//Put the Annotation on the method, but override the settings
@MyAnnotation(key="baz", value="buzz")
public void testValues() throws Exception {
    // Using reflection, get the public method "testValues", which is this method with no
args
    Method method = AnnotationExample.class.getMethod("testValues", null);

    // Fetch the Annotation that is of type MyAnnotation from the Method
    MyAnnotation annotation = (MyAnnotation)method.getAnnotation(MyAnnotation.class);

    // Print out the settings of the Annotation
    print(annotation);
}

public void print(MyAnnotation annotation) {
    // Fetch the MyAnnotation 'key' & 'value' properties, and print them out
    System.out.println(annotation.key() + " = " + annotation.value());
}

public static void main(String[] args) {
    AnnotationExample example = new AnnotationExample();
    try {
        example.testDefaults();
        example.testValues();
    } catch( Exception e ) {
        // Shouldn't throw any Exceptions
        System.err.println("Exception [" + e.getClass().getName() + "] - " +
e.getMessage());
        e.printStackTrace(System.err);
    }
}
}

```

はのようになります。

```

foo = bar
baz = buzz

```

をりす

Java 8までは、じアノテーションの2つのインスタンスを1つのにできませんでした。なは、のアノテーションのをするコンテナアノテーションをすることでした。

```

// Author.java
@Retention(RetentionPolicy.RUNTIME)
public @interface Author {
    String value();
}

```

```
// Authors.java
@Retention(RetentionPolicy.RUNTIME)
public @interface Authors {
    Author[] value();
}

// Test.java
@Authors({
    @Author("Mary"),
    @Author("Sam")
})
public class Test {
    public static void main(String[] args) {
        Author[] authors = Test.class.getAnnotation(Authors.class).value();
        for (Author author : authors) {
            System.out.println(author.value());
            // Output:
            // Mary
            // Sam
        }
    }
}
```

Java SE 8

Java 8では、`@Repeatable` アノテーションをして、コンテナアノテーションをよりクリーンでなで
`@Repeatable` ます。まずこれを `Author` クラスにします

```
@Repeatable(Authors.class)
```

これは、の `@Author` アノテーションを `@Authors` コンテナでまれているかのようにうようJavaにしま
 ず。また、`Class.getAnnotationsByType()` をして、`@Author` にそのコンテナではなくのクラスでアク
 セスすることもできます。

```
@Author("Mary")
@Author("Sam")
public class Test {
    public static void main(String[] args) {
        Author[] authors = Test.class.getAnnotationsByType(Author.class);
        for (Author author : authors) {
            System.out.println(author.value());
            // Output:
            // Mary
            // Sam
        }
    }
}
```

された

デフォルトでは、クラスアノテーションはそれらをするにはされません。これは、`@Inherited` ア
 ノテーションをアノテーションにすることでできます

の2つのアノテーションをえてみましょう。

```
@Inherited
@Target (ElementType.TYPE)
@Retention (RetentionPolicy.RUNTIME)
public @interface InheritedAnnotationType {
}
```

そして

```
@Target (ElementType.TYPE)
@Retention (RetentionPolicy.RUNTIME)
public @interface UninheritedAnnotationType {
}
```

3つのクラスにのようになげられているとします。

```
@UninheritedAnnotationType
class A {
}

@InheritedAnnotationType
class B extends A {
}

class C extends B {
}
```

このコードをする

```
System.out.println(new A().getClass().getAnnotation(InheritedAnnotationType.class));
System.out.println(new B().getClass().getAnnotation(InheritedAnnotationType.class));
System.out.println(new C().getClass().getAnnotation(InheritedAnnotationType.class));
System.out.println("_____");
System.out.println(new A().getClass().getAnnotation(UninheritedAnnotationType.class));
System.out.println(new B().getClass().getAnnotation(UninheritedAnnotationType.class));
System.out.println(new C().getClass().getAnnotation(UninheritedAnnotationType.class));
```

アノテーションのパッケージにじて、これにたがされます。

```
null
@InheritedAnnotationType()
@InheritedAnnotationType()

-----
@UninheritedAnnotationType()
null
null
```

は、インタフェースではなくクラスからできることにしてください。

アノテーション・プロセッサをしたコンパイル

このは、きのコンパイルチェックをうをしています。

アノテーション

`@Setter` アノテーションは、メソッドにマーカをすることができます。は、にされ、そのはできなくなります。

```
package annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.METHOD)
public @interface Setter {
}
```

アノテーションプロセッサ

`SetterProcessor` クラスは、アノテーションをするためにコンパイラによってされます。 `@Setter` アノテーションでをけられたメソッドが `public` で、 `set` でまりが4のの `static` メソッドであるかどうかをチェックします。これらのの1つがたされない、エラーが `Messenger` きまれます。コンパイラはこれを `stderr` にきみますが、のツールではこれを々にできます。たとえば `NetBeans IDE` では、エディタにエラーメッセージをするためにされるアノテーションプロセッサをユーザーができます。

```
package annotation.processor;

import annotation.Setter;
import java.util.Set;
import javax.annotation.processing.AbstractProcessor;
import javax.annotation.processing.Messenger;
import javax.annotation.processing.ProcessingEnvironment;
import javax.annotation.processing.RoundEnvironment;
import javax.annotation.processing.SupportedAnnotationTypes;
import javax.annotation.processing.SupportedSourceVersion;
import javax.lang.model.SourceVersion;
import javax.lang.model.element.Element;
import javax.lang.model.element.ElementKind;
import javax.lang.model.element.ExecutableElement;
import javax.lang.model.element.Modifier;
import javax.lang.model.element.TypeElement;
import javax.tools.Diagnostic;

@SupportedAnnotationTypes({"annotation.Setter"})
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class SetterProcessor extends AbstractProcessor {

    private Messenger messenger;

    @Override
```

```

    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv)
    {
        // get elements annotated with the @Setter annotation
        Set<? extends Element> annotatedElements =
roundEnv.getElementsAnnotatedWith(Setter.class);

        for (Element element : annotatedElements) {
            if (element.getKind() == ElementKind.METHOD) {
                // only handle methods as targets
                checkMethod((ExecutableElement) element);
            }
        }

        // don't claim annotations to allow other processors to process them
        return false;
    }

    private void checkMethod(ExecutableElement method) {
        // check for valid name
        String name = method.getSimpleName().toString();
        if (!name.startsWith("set")) {
            printError(method, "setter name must start with \"set\"");
        } else if (name.length() == 3) {
            printError(method, "the method name must contain more than just \"set\"");
        } else if (Character.isLowerCase(name.charAt(3))) {
            if (method.getParameters().size() != 1) {
                printError(method, "character following \"set\" must be upper case");
            }
        }
    }

    // check, if setter is public
    if (!method.getModifiers().contains(Modifier.PUBLIC)) {
        printError(method, "setter must be public");
    }

    // check, if method is static
    if (method.getModifiers().contains(Modifier.STATIC)) {
        printError(method, "setter must not be static");
    }
}

private void printError(Element element, String message) {
    messenger.printMessage(Diagnostic.Kind.ERROR, message, element);
}

@Override
public void init(ProcessingEnvironment processingEnvironment) {
    super.init(processingEnvironment);

    // get messenger for printing errors
    messenger = processingEnvironment.getMessenger();
}
}

```

パッケージング

コンパイラによってされるには、プロセッサをSPIでにすることがあります [ServiceLoader](#)を。

これをうには、`META-INF/services/javax.annotation.processing.Processor`というテキストファイルを、のファイルにえてアノテーションプロセッサとをむjarファイルにするがあります。ファイルにはプロセッサのをめるがあります。つまり、のようになります

```
annotation.processor.SetterProcessor
```

では、jarファイルが`AnnotationProcessor.jar`とばれるものとします。

アノテーションきクラスの

のクラスは、ポリシーによってしいにアノテーションがされている、デフォルトパッケージのサンプルクラスです。しかしながら、プロセッサのみが2のをなターゲットとみなす。

```
import annotation.Setter;

public class AnnotationProcessorTest {

    @Setter
    private void setValue(String value) {}

    @Setter
    public void setString(String value) {}

    @Setter
    public static void main(String[] args) {}

}
```

javacでアノテーションプロセッサをする

プロセッサがSPIをしてされた、プロセッサはきをするためににされます。えば
`AnnotationProcessorTest`クラスをコンパイルする

```
javac -cp AnnotationProcessor.jar AnnotationProcessorTest.java
```

のがられます

```
AnnotationProcessorTest.java:6: error: setter must be public
    private void setValue(String value) {}
        ^
AnnotationProcessorTest.java:12: error: setter name must start with "set"
    public static void main(String[] args) {}
        ^
2 errors
```

にコンパイルするわりに。 `.class` ファイルはされません。

これは、 `javac -proc:none` オプションをすることでぐことができます。わりに、 `-proc:only` するこ

とで、のコンパイルを`-proc:only`することもできます。

IDE

ネットビーンズ

プロセッサは、NetBeansエディタでできます。これをうには、プロセッサをプロジェクトでするがあります。

1. Project Properties > Build > Compiling
2. Enable Annotation ProcessingをEnable Annotation ProcessingをEnable Annotation Processing in EditorためのチェックマークをEnable Annotation Processing in Editorする
3. アノテーションプロセッサリストのにある[Add] Addクリックします
4. されるポップアップで、プロセッサのクラスをし、[Ok]をクリックします。

```
1  import annotation.Setter;
2
3  public class Annotation {
4      @Setter
5      private void setValue(String value) {}
6
7      @Setter
8      public void setString(String value) {}
9
10     @Setter
11     public static void main(String[] args) {}
12
13 }
14
15
```

setter must be public

(Alt-Enter shows hints)

のにあるアイデア

Javaでは、についてのようになっています。

アノテーションは、をプログラムにけるマーカーですが、にはしません。

は、またはのにれます。やのにできるにすることはです。

がにされるものは、「メタ」`@Target`によってされます。は、「アノテーションの」をしてください。

はさまざまなでされます。SpringやSpring-MVCのようなフレームワークでは、アノテーションをして、をするやをルーティングするをします。

のフレームワークでは、アノテーションをコードにしています。LombokとJPAはJavaおよび

SQLコードをするアノテーションをするなです。

このトピックでは、のについてにします。

- のをする
- Javaがするはですか
- にはどのようにされますか

「this」とレシーバパラメータのアノテーション

Javaアノテーションがめてされたとき、インスタンスメソッドのターゲットにをけるためのや、クラスコンストラクタのしコンストラクタパラメータはありませんでした。これは、レシーバパラメータをしてJava 8でされました。 [JLS 8.4.1](#)をしてください。

receiverパラメータは、インスタンスメソッドまたはクラスのコンストラクタのためのオプションのデバイスです。インスタンスメソッドの、receiverパラメータは、メソッドがびされるオブジェクトをします。クラスのコンストラクタの、receiverパラメータはしくされたオブジェクトのすぐにまれたインスタンスをします。どちらのでも、レシーバパラメータは、されたオブジェクトのをソースコードにすことができるだけであり、そのにをけることができます。パラメータはパラメータではありません。よりには、のではなく4.12.3、メソッドびしやされたクラスインスタンスでとしてされるにしてされず、。

のは、ののレシーバパラメータのをしています。

```
public class Outer {
    public class Inner {
        public Inner (Outer this) {
            // ...
        }
        public void doIt (Inner this) {
            // ...
        }
    }
}
```

レシーバパラメータのは、アノテーションをできるようにすることです。たとえば、メソッドのびしにCloseableオブジェクトがじられていないことをすることをとするカスタム@IsOpenがあるとします。えば

```
public class MyResource extends Closeable {
    public void update(@IsOpen MyResource this, int value) {
        // ...
    }

    public void close() {
        // ...
    }
}
```



```
}
```

1つのレベルでは、`@IsOpen`の`this`にドキュメントとしてちます。しかし、々はにもっとくをうことができます。えは

- プロセッサは、ことのランタイムチェックをすることができ`this`でない`update`ばれています。
- コードチェッカーは、コードをして、`update`がびされたときに`this`をじることができるケースをつけることができます。

のをする

`Annotation`パラメータは、としてされているはのをけることができます。えは、アノテーション`@SuppressWarnings`はのようになれます。

```
public @interface SuppressWarnings {  
    String[] value();  
}
```

`value`パラメータはのです。にたをして、のをできます。

```
@SuppressWarnings({"unused"})  
@SuppressWarnings({"unused", "javadoc"})
```

のだけをするがあるは、をすることができます。

```
@SuppressWarnings("unused")
```

オンラインでをむ <https://riptutorial.com/ja/java/topic/157/>

174: なインターフェイス

Fluent Interfaceのなはみやすさのです。

オブジェクトをするためにされる、びしができるは、コンパイルのチェックをしてにわれ、されます。たとえば、なオブジェクトをするためのパスにったステップをすのオプションのツリーをえてみましょう。

```
A -> B
  -> C -> D -> Done
      -> E -> Done
      -> F -> Done.
      -> G -> H -> I -> Done.
```

なインターフェースをするビルダーは、びしがステップでなオプションをにできるようになります。たとえば、**A→B**でもですが、**A→C**ではなく、コンパイルにエラーがします。

Examples

なテストフレームワーク

「のい」から<http://google.github.io/truth/>

```
String string = "awesome";
assertThat(string).startsWith("awe");
assertWithMessage("Without me, it's just aweso").that(string).contains("me");

Iterable<Color> googleColors = googleLogo.getColors();
assertThat(googleColors)
    .containsExactly(BLUE, RED, YELLOW, BLUE, GREEN, RED)
    .inOrder();
```

なプログラミングスタイル

なプログラミングスタイルでは、でないプログラミングスタイルではもさないなセッターメソッドから`this`をします。

これにより、さまざまなメソッドびしをさせることができ、コードをくしてにとっていやすくします。

このなコードをえてみましょう

```
public class Person {
    private String firstName;
    private String lastName;

    public String getFirstName() {
        return firstName;
    }
}
```

```

    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String whoAreYou() {
        return "I am " + firstName + " " + lastName;
    }

    public static void main(String[] args) {
        Person person = new Person();
        person.setFirstName("John");
        person.setLastName("Doe");
        System.out.println(person.whoAreYou());
    }
}

```

セッターメソッドはもさないので、いくつかのデータで `Person` をインスタンスしてするには、`main` メソッドで4つのがです。なスタイルでこのコードをすることができます

```

public class Person {
    private String firstName;
    private String lastName;

    public String getFirstName() {
        return firstName;
    }

    public Person withFirstName(String firstName) {
        this.firstName = firstName;
        return this;
    }

    public String getLastName() {
        return lastName;
    }

    public Person withLastName(String lastName) {
        this.lastName = lastName;
        return this;
    }

    public String whoAreYou() {
        return "I am " + firstName + " " + lastName;
    }

    public static void main(String[] args) {
        System.out.println(new Person().withFirstName("John")
            .withLastName("Doe").whoAreYou());
    }
}

```

```
}
```

アイデアは、メソッドコールチェーンのとなをするメソッドのをにするために、にいくつかのオブジェクトをすことです。このなスタイルは、コードをみやすくします。

オンラインでなインターフェイスをむ <https://riptutorial.com/ja/java/topic/5090>/なインターフェイス

175:

き

Javaプログラミングのは、1つ、2つ、または3つのオペランドでのをし、をすなシンボルです。

オペレータは、1つ、2つまたは3つのオペランドでをするJavaプログラムをえるシンボルはシンボルです。とそのオペランドがをします のトピックを。のオペランドは、それがです。

このトピックでは、Javaでされた40のなるについてします。したのトピックでは、

- 、オペランドなどがにどのようにみわされているか、
- の、および
- の、、およびのみ

Examples

+

`+`は、Javaの3つのなるをします。

- `+`にオペランドがないは、それはプラスです。
- 2つのオペランドがあり、それらがともである。それはバイナリです。
- 2つのオペランドがあり、それらのうちのなくとも1つがStringである、それは2です。

なケースでは、は2つのをして3のをします。えば

```
String s1 = "a String";
String s2 = "This is " + s1;    // s2 contains "This is a String"
```

2つのオペランドの1つがでない、のようにStringされます。

- がプリミティブのオペランドは、boxedの`toString()`をびすことによってされます。
- かのオペランドは、オペランドの`toString()`メソッドをびすことによってされます。オペランドが`null`、または`toString()`メソッドが`null`すは、わりにリテラル`"null"`がされます。

えば

```
int one = 1;
String s3 = "One is " + one;    // s3 contains "One is 1"
String s4 = null + " is null";  // s4 contains "null is null"
String s5 = "{1} is " + new int[]{1}; // s5 contains something like
                                     // "{1} is [I@xxxxxxxxx]"
```

s5のは、のtoString()メソッドがjava.lang.Objectからされ、そのはとオブジェクトのIDハッシュコードからなるをすることです。

がのをいて、をしてしいStringオブジェクトをします。の、はコンパイル・タイプでされ、ランタイムはリテラルとです。これは、いリテラルをのようにするにランタイムオーバーヘッドがないことをします。

```
String typing = "The quick brown fox " +
    "jumped over the " +
    "lazy dog";           // constant expression
```

と

のように、をいて、はしいStringオブジェクトをします。このコードをえてみましょう

```
public String stars(int count) {
    String res = "";
    for (int i = 0; i < count; i++) {
        res = res + "*";
    }
    return res;
}
```

のメソッドでは、ループのりしは、のりしより1いしいStringをします。は、しいStringをするためにオペランドストリングのすべてのをコピーします。したがって、stars(N)は

- NしいStringオブジェクトをし、のものをくすべてのオブジェクトをStringます。
- $N * (N + 1) / 2$ をコピーし、
- ゴミの $O(N^2)$ バイトをします。

これはきなNにしてにです。、ループでをするコードでは、このがするがあります。これをくためのよりいはのようになります

```
public String stars(int count) {
    // Create a string builder with capacity 'count'
    StringBuilder sb = new StringBuilder(count);
    for (int i = 0; i < count; i++) {
        sb.append("*");
    }
    return sb.toString();
}
```

には、にするがありStringBuilder、これはではない、クラスがにビルダーがをするためにするをします。このでは、 $O(N^2)$ ではなく $O(N)$ にのコピーがされます。

このパターンをすべてのにするもいます。ただし、JLSではJavaコンパイラがののができるため、これはです。えば

```
String s1 = ...;
```

```
String s2 = ...;
String test = "Hello " + s1 + ". Welcome to " + s2 + "\n";
```

、バイトコードコンパイラによってこのようなものにされます。

```
StringBuilder tmp = new StringBuilder();
tmp.append("Hello ")
tmp.append(s1 == null ? "null" + s1);
tmp.append("Welcome to ");
tmp.append(s2 == null ? "null" + s2);
tmp.append("\n");
String test = tmp.toString();
```

JITコンパイラは、`s1`または`s2`が`null`はならないとできるには、それをさらにすることができます。しかし、このはののみです。

するに、のをしている、

- ループで`is`をっているは、をしてくださいまたはしたもの。
- のをでしないでください。

`+`、`-`、`*`、`/`、

Javaには、とのをする7つのがされています。

- `+`は2つあります。
 - `2`は、1つのをのにします。をするバイナリ`+`もあります。のでします。
 - プラスは、のトリガーとはにもしません
- 2つのがあります`-`は、
 - バイナリは、のから1つのをします。
 - マイナスは、そのオペランドをゼロからするのとです。
- バイナリ`*`は、あるをのにします。
- `2/`は、あるをのでします。
- バイナリ`1`は、あるをのでったときのをします。

これはしばしば「モジュラス」とってばれます。「`1`」は、JLSによってされるです。"Modulus"と "remainder"はじものではありません。

オペランドとの、および

はオペランドをとし、をします。オペランドのはのプリミティブなつまり、`byte`、`short`、`char`、`int`、`long`、`float`または`double` または `java.lang` されたのラッパーです。すなわち、`Byte`、`Character`、`Short`、`Integer`、`Long`、`Float` または `Double`。

のは、のように、オペランドまたはオペランドのについてされます。

- いずれかのオペランドが`double`または`Double`、のは`double`ます。

- それの、どちらかのオペランドが`float`または`Float`、のは`float`です。
- それの、オペランドのいずれかが`long`または`Long`、のは`long`です。
- それの、のは`int`です。これは`byte`、`short`、および`char`オペランドと`int`。

ののによって、のと、そのオペランドのがまります

- のである`double`、オペランドにさ`double`、およびは、64ビットバイナリIEEE 754をいてわれます。
- のである`float`、オペランドがにされた`float`、びは、32ビットバイナリIEEE 754をいてわれます。
- のタイプがある`long`、オペランドにされた`long`、およびは、64ビットのき2のバイナリをいてわれます。
- のである`int`、オペランドにされている`int`、およびは、32ビットき2のバイナリをいてわれます。

プロモーションは2つのでされます。

- オペランド・タイプがラッパー・タイプの、オペランドはするプリミティブ・タイプののにアンボックスされます。
- にじて、プリミティブをなにさせます。
 - するの`int`または`long`ロスレスです。
 - `float double`はです。
 - をにすると、がするがあります。は、IEEE 768「もいめ」セマンティクスをしてされます。

の

l は、のオペランド n とのオペランド d をし、 q をします。

Javaのはゼロにかってめます。 [JLSセクション15.17.2](#)では、Javaのるいをのようになっています。

オペランド n および d についてされたは、 $|d \cdot q| \leq |n|$ をしながらなりきさの q である $|d \cdot q| \leq |n|$ 。また、 q ときにである $|n| \geq |d|$ そして n び d じをとっていますが、 q のとき $|n| \geq |d|$ n と d はのがあります。

なケースが2つあります。

- n が`MIN_VALUE`でが-1の、オーバーフローがし、は`MIN_VALUE`ます。この、はスローされません。
- d が0の、`ArithmeticException`がスローされます。

Javaのでは、よりくのケースをするがあります。しかし、なえは、 q が $d \cdot q = n$ をたすのにもいであるということです $d \cdot q = n$ 。

のでがすることはありません。わりに、ゼロでするは、`INF`と`NaN`のになります。。

の

CおよびC++とはなり、Javaのりのは、とのでします。

のについての $a \% b$ であるとされる r のように $(a / b) * b + r$ にしく、 $a /$ 、 $*$ および $+$ なJavaです。これは、 b が0のをいて、すべてののにされます。その、は`ArithmeticException`ます。

これは、のから、の $a \% b$ のみであることができるであり、のみです。 a また、きさの $a \% b$ にのきさである b 。

のは、ののです。 $a \% b$ の r が $r = a - (b \cdot q)$ でされます。ここで、

- q はであり、
- a / b がのにのみであり、 a / b がであるのみであり、
- そのきさは、のなのきえることなく、なりきいと $a b$ 。

のは、 b がゼロのときのように、エッジケースで`INF`と`NaN`をできます。。をスローしません。

な

$\%$ でされたのは、IEEE 754でされたによってされたものとはではありません。IEEE 754は、`Math.IEEEremainder` ライブラリメソッドをしてできます。

オーバーフロー

Java 32および64ビットのはきで、2の2をします。えは、のは、32ビットとして`int 1 - 31 -2 +2`スル—31。

2つの N ビット $N = 32$ または 64 を、、またはすると、がきすぎて N ビットとしてすことができません。この、はオーバーフローにつながり、はのようになります。

- は、の2のをえるためにされる。このは N ビットよりきい。
- の32または64ビットがとしてされます。

オーバーフローはどのようなでもにならないことにしてください。

のINFとNaN

Javaは`float`と`double` IEEE 754をします。これらのには、のにあるをすためのながあります。

- 「」または`INF`は、きすぎるをします。`+INF`は、きすぎてのをします。`-INF`は、きすぎてのをします。
- 「」/「ではない」または`NaN`はなからじるをします。

`INF`は、オーバーフローをきこす、またはゼロによってされます。

NaNは、ゼロをゼロとするか、ゼロをゼロすることによってされます。

くべきことに、をトリガーせずにINFおよびNaNオペランドをしてをすることはです。例えば

- + INFとのをえると+ INFがられます。
- + INFと+ INFをすると+ INFがられます。
- + INFと-INFをするとNaNがられます。
- INFですと、+0.0または-0.0がられます。
- 1つまたはのNaNオペランドをするすべてのNaNがされます。

については、 [JLS 15](#)のするサブセクションをしてください。これはに「」であることにしてください。なの、 `INF`または`NaN`はかがっていることをします。たとえば、データがであるかっているか、がってプログラムされているなどです。

==、 =

==と!=は、オペランドがしいかどうかによって`true`または`false`とされる2です。==は、オペランドがしいは`true ==`、そうでないは`false`ます。!=は、オペランドがしいは`false`し、それのは`true`します。

これらはプリミティブとのオペランドでできますが、はきくなります。JLSによると、これらのはにはには3つのなるセットがあります。

- Boolean ==および!=。
- ==および!=。
- ==および!=。

ただし、すべてのにおいて、 ==および!=のは`boolean`です。

==および!=

==または!=のオペランドのまたはがプリミティブな `byte`、 `short`、 `char`、 `int`、 `long`、 `float`または`double`である、はです。2のオペランドは、プリミティブなまたはボックスされたのいずれかでなければなりません。

ののはのとおりです。

1. オペランドの1つがボックスであれば、ボックスされません。
2. いずれかのオペランドが`byte`、 `short`または`char`、 `int`されます。
3. オペランドのがじでない、「よりさい」のオペランドは「よりきい」にされる。
4. はのようにされます。
 - プロモートされたオペランドが`int`または`long`、がしているかどうかテストされます。
 - プロモートされたオペランドが`float`または`double`は、のようになります。
 - 0の2つのバージョン `+0.0`と`-0.0` はしいと`-0.0`れます
 - `NaN`はにもしくないとしてわれます。

- 。 IEEE 754がであれば、のはしくなります。

==と!=をしてをするはがです。

ブール==および!=

のオペランドが`boolean`であるか、またはが`boolean`で、が`Boolean`、これらのはブール==および!=です。はのとおりです。

1. オペランドの1つが`Boolean`、ボックスはアンボックスされます。
2. ボックスされていないオペランドがテストされ、ブールがのによってされます

A	B	A == B	A = B

で==と!=えめにすることをおめする2つの「とし」があります

- ==または!=をして2つの`Boolean`オブジェクトをすると、がされます。しないがじることがあります。 [Pitfall==をしてIntegerなどのプリミティブラッパオブジェクトをする](#)
- ==はに=とえることがあります。ほとんどのオペランドタイプでは、このエラーはコンパイルエラーにつながります。しかし、`boolean`オペランドと`Boolean`オペランドのいは、ったランタイムにつながります。 [としをる-'=='をとってブールをテストする](#)

==および!=

のオペランドがオブジェクトである、==と!=は、2つのオペランドがじオブジェクトをしているかどうかをテストします。これはしばしばあなたがむものではありません。2つのオブジェクトがじであるかどうかをべるには、わりに`.equals()`メソッドをするがあります。

```
String s1 = "We are equal";
String s2 = new String("We are equal");

s1.equals(s2); // true

// WARNING - don't use == or != with String values
s1 == s2;      // false
```

`String`をするために==および!=をすると、ほとんどのしくありません。

<http://www.riptuit.com/java/example/16290/pitfall--using-to-compare-strings>をしてください。じようながプリミティブラッパにもてはまります。 <http://www.riptuit.com/java/example/8996/pitfall--using-to-compare-primitive-wrappers-objects-such-as-integer>をしてください。

NaNのエッジケースについて

JLS 15.21.1は、のようにべています。

いずれかのオペランドがNaN、==のはfalseが、!=のはtrueです。、テスト `x != x` ある `true` のに `x` があるNaN。

これは、ほとんどのプログラマにとってしないものです。NaNがそれとしいかどうかをテストすると、えは "いいえ、そうではありません" です。いえれば、==はNaNにしてではありません。

しかし、これはJavaの「」ではなく、これはIEEE 754でされており、のプログラミングでされています。については、<http://stackoverflow.com/a/1573715/139985>をしてください...これは、「がされたときにいた」がいたことにしてください

インクリメント/デクリメント `++ / -`

`++`および`--`をして、を1ずつインクリメントまたはデクリメントできます。

`++`と`--`がにうとき、それはポスト インクリメントとポストデクリメントとばれます。

```
int a = 10;
a++; // a now equals 11
a--; // a now equals 10 again
```

`++`と`--`がのにあるとき、はそれぞれプリ インクリメントとプリデクリメントとばれます。

```
int x = 10;
--x; // x now equals 9
++x; // x now equals 10
```

がにする、のはインクリメントまたはデクリメントされたのです。がのにく、のはインクリメントまたはデクリメントされるのです。

```
int x=10;

System.out.println("x=" + x + " x=" + x++ + " x=" + x); // outputs x=10 x=10 x=11
System.out.println("x=" + x + " x=" + ++x + " x=" + x); // outputs x=11 x=12 x=12
System.out.println("x=" + x + " x=" + x-- + " x=" + x); // outputs x=12 x=12 x=11
System.out.println("x=" + x + " x=" + --x + " x=" + x); // outputs x=11 x=10 x=10
```

ポストインクリメントまたはデクリメントをきしないようにしてください。これは、イン/デクリメントされたにりてされるのにポストイン/デクリメントをするにします。イン/デクリメントはをちません。のがしくインクリメントされても、そののはのからにされたでちにきされます

```
int x = 0;
x = x++ + 1 + x++;           // x = 0 + 1 + 1
                             // do not do this - the last increment has no effect (bug!)
System.out.println(x);      // prints 2 (not 3!)
```

しい

```
int x = 0;
x = x++ + 1 + x;      // evaluates to x = 0 + 1 + 1
x++;                  // adds 1
System.out.println(x); // prints 3
```

き:)

{をする} {ステートメントがに`true`} {ステートメントがに`false`}

にされているように、1ともばれるは? と:コロンをして、2つのなのをにします。これは、よりい
if-else ブロックをきえて、にづいて2つのの1つをすためにできます。

```
result = testCondition ? value1 : value2
```

です

```
if (testCondition) {
    result = value1;
} else {
    result = value2;
}
```

「testConditionが`true`のは、を`value1`にします。それのは、result2を`value2`にします。

えば

```
// get absolute value using conditional operator
a = -10;
int absValue = a < 0 ? -a : a;
System.out.println("abs = " + absValue); // prints "abs = 10"
```

です

```
// get absolute value using if/else loop
a = -10;
int absValue;
if (a < 0) {
    absValue = -a;
} else {
    absValue = a;
}
System.out.println("abs = " + absValue); // prints "abs = 10"
```

な

キヌルチェックなどにをできます。

```
String x = y != null ? y.toString() : ""; //where y is an object
```

このはのものとです。

```
String x = "";

if (y != null) {
    x = y.toString();
}
```

は、2にいするのでを、そこりのはほとんどありませんが、は、のとみわせたにのりとされます。

```
// no parenthesis needed for expressions in the 3 parts
10 <= a && a < 19 ? b * 5 : b * 7

// parenthesis required
7 * (a > 0 ? 2 : 5)
```

のネストは、3のでうこともできます.3のでは、チェーンやswitchのようにします。

```
a ? "a is true" :
b ? "a is false, b is true" :
c ? "a and b are false, c is true" :
    "a, b, and c are false"

//Operator precedence can be illustrated with parenthesis:

a ? x : (b ? y : (c ? z : w))
```

1 - [JavaとJavaチュートリアル](#)ので ? : をとびます。このチュートリアルでは、Javaでされているのであるため、「」ともばれています。「き」のは、のをするCおよびC ++およびのとしています。

ビットと、 、 |、 ^

Javaでは、オペランドまたはブールオペランドにしてビットまたはをする4つのがされています。

- ~ は、1つのオペランドのビットのビットまたはをするです。 [JLS 15.15.5](#)をしてください。
- AND & は、2つのオペランドのビットまたははな "and"をする2です。 [JLS 15.22.2](#)をしてください。
- OR | は2つのオペランドのビットまたははな "または"をする2です。 [JLS 15.22.2](#)をしてください。
- XOR ^ は、2つのオペランドのビットまたは「」をする2です。 [JLS 15.22.2](#)をしてください。

オペランドがブールのかにこれらのによってされるは、のようになえます。

A	B	A	AB	B	A ^ B
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

オペランドの、のは々のビットについてがくるかをしていることにしてください。には、32ビットまたは64ビットのオペランドまたはオペランドのすべてをにします。

オペランドのとの。

のは、オペランドがのにされます。ビットのな

~は、ブールをするために、またはオペランドのすべてのビットをするためにされます。

&は、オペランドのビットのを「マスクする」ためにされます。えは

```
int word = 0b00101010;  
int mask = 0b00000011;    // Mask for masking out all but the bottom  
                           // two bits of a word  
int lowBits = word & mask;    // -> 0b00000010  
int highBits = word & ~mask;  // -> 0b00101000
```

|は、2つのオペランドのをするためにされます。えは

```
int word2 = 0b01011111;  
// Combine the bottom 2 bits of word1 with the top 30 bits of word2  
int combined = (word & mask) | (word2 & ~mask);    // -> 0b01011110
```

^は、ビットをトグルまたはフリップするためにされます。

```
int word3 = 0b00101010;  
int word4 = word3 ^ mask;    // -> 0b00101001
```

ビットのについては、「ビット」をしてください。

オペレータのインスタンス

これは、オブジェクトがのクラス/インタフェースであるかどうかをチェックします。 instanceof はのようになえます。

```
( Object reference variable ) instanceof (class/interface type)
```

```
public class Test {  
  
    public static void main(String args[]){  
        String name = "Buyya";  
        // following will return true since name is type of String  
        boolean result = name instanceof String;  
        System.out.println( result );  
    }  
}
```

これにより、のがられます。

```
true
```

されるオブジェクトがのとのあるである、このはききをします。

```
class Vehicle {}  
  
public class Car extends Vehicle {  
    public static void main(String args[]){  
        Vehicle a = new Car();  
        boolean result = a instanceof Car;  
        System.out.println( result );  
    }  
}
```

これにより、のがられます。

```
true
```

$=$ 、 $+$ 、 $-$ 、 $*$ 、 $/$ 、 $=$ 、 $<<=$ 、 $>>=$ 、 $>>>=$ 、 $=$ 、 $|=$ および $=$

これらのオペランドは、でものでもなければなりません。オペランドはオペランドとりてがなければなりません。これは、がじでなければならぬか、またはオペランドがオペランドにボックス、アンボックスまたはのみわせによってでなければならぬことをします。については、[JLS 5.2](#)をしてください。

「と」のなは、[JLS 15.26.2](#)のようにされています。

フォームの $E_1 \text{ op } E_2$ とである $E_1 = (T) \ ((E_1 \text{ op } (E_2)) \ T)$ タイプである E_1 ことをいて、 E_1 だけされます。

なりてのにのキャストがあることにしてください。

1. $=$

オペランドのをオペランドにします。

$c = a + b$ のをする $a + b$ に c とにりて c

2. +=

"add and assign" オペランドのにオペランドのをし、オペランドにをする。オペランドが `String` であれば、これは "してする" です。

$c += a$ は $c = c + a$ とおおよそじです。

3 -=

"subtract and assign" は、オペランドのからをし、をオペランドにします。

$c -= a$ は $c = c - a$ とおおよそじです

4. *=

"と" オペランドのにオペランドのをし、そのをオペランドにする。。

$c *= a$ は $c = c * a$ とおおよそじです

5. /=

「と」オペランドのをオペランドのでし、そのをオペランドにする。

$c /= a$ は、 $c = c / a$ とおおよそじです。

6. %=

"modulus and assign" オペランドののモジュラスをオペランドのでし、そのをオペランドにする。

$c \%*= a$ は、 $c = c \% a$ とおおよそじです

7. <=

"left shift and assign"。

$c <= 2$ は、 $c = c << 2$ とおおよそじです。

>>=

"シフトと"。

$c >>= 2$ は $c = c >> 2$ とおおよそじです

>>>=

"シフトとりて"。

$c >>>= 2$ は、 $c = c >>> 2$ とおおよそじです。

&=

"bitwise and assign".

$c \&= 2$ は $c = c \& 2$ とおおよそ同じです

||

"bitwise or assign".

$c |= 2$ は $c = c | 2$ とほぼ同じ $c = c | 2$

12. ^=

「ビットごとのandおよび」。

$c \wedge= 2$ は、 $c = c \wedge 2$ とおおよそ同じです

きおよびきまたは**&&**および**||**は、

Javaにはきのきときのがされており、いずれも `boolean` の1つまたは2つのオペランドをとり、`boolean` をします。これらは

- **&&** - きAND、
- **||** - きOR。 `<left-expr> && <right-expr>` のは、のコードにします。

```
{
    boolean L = evaluate(<left-expr>);
    if (L) {
        return evaluate(<right-expr>);
    } else {
        // short-circuit the evaluation of the 2nd operand expression
        return false;
    }
}
```

`<left-expr> || <right-expr>` の `<left-expr> || <right-expr>` は、のコードにします。

```
{
    boolean L = evaluate(<left-expr>);
    if (!L) {
        return evaluate(<right-expr>);
    } else {
        // short-circuit the evaluation of the 2nd operand expression
        return true;
    }
}
```

のコードがすように、のは `if / else` をすることと `else` 。

- &&をのガードとしてする

のは、&&のもなパターンをしています。これらの2つのバージョンのメソッドをして、されたIntegerがゼロかどうかをテストします。

```
public boolean isZero(Integer value) {
    return value == 0;
}

public boolean isZero(Integer value) {
    return value != null && value == 0;
}
```

ほとんどの、のバージョンがしますが、valueがnull、NullPointerExceptionがスローされます。

2のバージョンでは、「ガード」テストをしました。value != null && value == 0は、にvalue != null testをするvalue != nullによってされvalue != null。nullそれがされ、すなわち、テストがしたtrue、そのvalue == 0がされます。nullがした、value == 0はスキップされ、NullPointerExceptionはNullPointerException しません。

- コストのかかるをけるために&&をする

のは、&&をしてなをけるをしています。

```
public boolean verify(int value, boolean needPrime) {
    return !needPrime | isPrime(value);
}

public boolean verify(int value, boolean needPrime) {
    return !needPrime || isPrime(value);
}
```

のバージョンでは、|にされるので、な isPrimeメソッドはにびされます。2のバージョンは、||をしてなびしをします。|わりに|。

シフト<<、>>および>>>

Javaは、32ビットおよび64ビットのをビットでシフトする3つのをします。これらはすべてバイナリであり、のオペランドはシフトされるであり、2のオペランドはどれだけシフトするかです。

- <<またはシフトは、1オペランドによってえられるを、2オペランドによってえられるビットのだけにシフトする。のはゼロでめられます。
- >>>またはシフトは、1オペランドによってえられるを、2オペランドによってえられるビットのだけにシフトする。のは、のビットをコピーすることによってめられます。このプロセスはとしてられています。

- '>>>'またはシフトは、1オペランドによってえられるを、2オペランドによってえられるビットのだけにシフトする。のは0でめられます。

ノート

1. これらの、1オペランドとして `int` または `long` をとし、1オペランドとじのをします。シフトのを `byte`、`short` または `char` にするときは、なキャストをするがあります。
2. のオペランドが `byte`、`char` または `short` であるシフトをすると、それは `int` され、は `int` します。
3. 2オペランドは、シフトをえるために、のビットをとしてじられる。 **mod** ののについては、[Modulus](#)のをしてください。
4. によってまたはからシフトされたビットはされます。Javaはプリミティブな "" をしません。
5. シフトは、2のを2のでしてです。
6. シフトは、aの2のに2のをけることとです。

のは、3つのシフトのをするのにちます。はをけるために2でされています。

オペランド1	オペランド2	<<	>>	>>>
0b00000000000001011	0	0b00000000000001011	0b00000000000001011	0b00000000000001011
0b00000000000001011	1	0b000000000000010110	0b0000000000000101	0b0000000000000101
0b00000000000001011	2	0b0000000000000101100	0b0000000000000010	0b0000000000000010
0b00000000000001011	28	0b1011000000000000	0b0000000000000000	0b0000000000000000
0b00000000000001011	31	0b1000000000000000	0b0000000000000000	0b0000000000000000
0b00000000000001011	32	0b00000000000001011	0b00000000000001011	0b00000000000001011
...	...	...	...	...
0b10000000000001011	0	0b10000000000001011	0b10000000000001011	0b10000000000001011
0b10000000000001011	1	0b000000000000010110	0b1100000000000101	0b0100000000000101
0b10000000000001011	2	0b0000000000000101100	0b1110000000000010	0b00100000000000100
0b10000000000001011	31	0b1000000000000000	0b1111111111111111	0b0000000000000001

ビットにおけるシフトの

ラムダ ->

Java 8では、Lambda -> はLambdaをするためにされるです。これらのにすように、のが2つあります。

Java SE 8

```
a -> a + 1           // a lambda that adds one to its argument
a -> { return a + 1; } // an equivalent lambda using a block.
```

ラムダは、またはよりには、インタフェースをするクラスのインスタンスをします。

このは、のためにここにまれています。ないについては、 [Lambda Expressions](#)のトピックをしてください。

<、<=、>、>=

<、<=、>および>=は、をするための2です。のはりです。たとえば、`a`と`b`が`byte`、`short`、`char`、`int`、`long`、`float`、`double`またはするボックスのいずれかとされている`a`

```
- `a < b` tests if the value of `a` is less than the value of `b`.
- `a <= b` tests if the value of `a` is less than or equal to the value of `b`.
- `a > b` tests if the value of `a` is greater than the value of `b`.
- `a >= b` tests if the value of `a` is greater than or equal to the value of `b`.
```

これらののは、すべてのにおいて`boolean`です。

をすると、なるタイプのをできます。えは

```
int i = 1;
long l = 2;
if (i < l) {
    System.out.println("i is smaller");
}
```

は、またはのがボックスされたのインスタンスであるにできます。えは

```
Integer i = 1; // 1 is autoboxed to an Integer
Integer j = 2; // 2 is autoboxed to an Integer
if (i < j) {
    System.out.println("i is smaller");
}
```

なはのようにされます。

1. オペランドの1つがボックスであれば、ボックスされません。
2. いずれかのオペランドが`byte`、`short`または`char`、`int`されます。
3. オペランドのがじでない、「よりさい」のオペランドは「よりきい」にされる。
4. はの`int`、`long`、`float`または`double`にしてされます。

をむにはがです。

- をするでは、コンピュータのがられているため、めがすることがよくあります。
- とをする、をにすると、めがするもあります。

に、Javaは、ののののをしサポートします。たとえば、これらのをして、のなどをすることはできません。

オンラインでをむ <https://riptutorial.com/ja/java/topic/176/>

176:

- `LocalTime time = LocalTime.now;` //のシステムクロックです
- `LocalTime time = LocalTime.MIDNIGHT;` // 00:00
- `LocalTime time = LocalTime.NOON;` // 12:00
- `LocalTime time = LocalTime.of(12,12,45);` // 12:12:45

パラメーター

<code>LocalTime.of(13, 12, 11)</code>	13:12:11
<code>LocalTime.MIDNIGHT</code>	00:00
<code>LocalTime.NOON</code>	12:00
<code>LocalTime.now()</code>	システムクロックからの
<code>LocalTime.MAX</code>	サポートされているの235959.999999999
<code>LocalTime.MIN</code>	サポートされているローカルの00:00
<code>LocalTime.ofSecondOfDay(84399)</code>	23:59:59、2のからを
<code>LocalTime.ofNanoOfDay(2000000000)</code>	00:00:02、ナノのからを

クラスがすように、`LocalTime`はのなをします。それはをすものではありません。それはえられたのなラベルです。

クラスはベースであり、をうときは`equals`メソッドをするがあります。

このクラスは`java.time`パッケージからのものです。

Examples

の

、、、およびナノをできます。

```
LocalTime time = LocalTime.now();
LocalTime addHours = time.plusHours(5); // Add 5 hours
LocalTime addMinutes = time.plusMinutes(15) // Add 15 minutes
LocalTime addSeconds = time.plusSeconds(30) // Add 30 seconds
LocalTime addNanoseconds = time.plusNanos(150_000_000) // Add 150.000.000ns (150ms)
```

とその

```
import java.time.LocalDateTime;
import java.time.ZoneId;
import java.time.temporal.ChronoUnit;

public class Test {
    public static void main(String[] args)
    {
        ZoneId zone1 = ZoneId.of("Europe/Berlin");
        ZoneId zone2 = ZoneId.of("Brazil/East");

        LocalDateTime now = LocalDateTime.now();
        LocalDateTime now1 = LocalDateTime.now(zone1);
        LocalDateTime now2 = LocalDateTime.now(zone2);

        System.out.println("Current Time : " + now);
        System.out.println("Berlin Time : " + now1);
        System.out.println("Brazil Time : " + now2);

        long minutesBetween = ChronoUnit.MINUTES.between(now2, now1);
        System.out.println("Minutes Between Berlin and Brazil : " + minutesBetween
+"mins");
    }
}
```

2つのLocalTimeの

2つのLocalTimeののをするには、1 LocalDateTime.until(Temporal, TemporalUnit) メソッドと2 TemporalUnit.between(Temporal, Temporal) 2つののがあります。

```
import java.time.LocalDateTime;
import java.time.temporal.ChronoUnit;

public class AmountOfTime {

    public static void main(String[] args) {

        LocalDateTime start = LocalDateTime.of(1, 0, 0); // hour, minute, second
        LocalDateTime end = LocalDateTime.of(2, 10, 20); // hour, minute, second

        long halfDays1 = start.until(end, ChronoUnit.HALF_DAYS); // 0
        long halfDays2 = ChronoUnit.HALF_DAYS.between(start, end); // 0

        long hours1 = start.until(end, ChronoUnit.HOURS); // 1
        long hours2 = ChronoUnit.HOURS.between(start, end); // 1

        long minutes1 = start.until(end, ChronoUnit.MINUTES); // 70
        long minutes2 = ChronoUnit.MINUTES.between(start, end); // 70

        long seconds1 = start.until(end, ChronoUnit.SECONDS); // 4220
        long seconds2 = ChronoUnit.SECONDS.between(start, end); // 4220

        long millisecs1 = start.until(end, ChronoUnit.MILLIS); // 4220000
        long millisecs2 = ChronoUnit.MILLIS.between(start, end); // 4220000

        long microsecs1 = start.until(end, ChronoUnit.MICROS); // 4220000000
    }
}
```



```

    long microsecs2 = ChronoUnit.MICROS.between(start, end); // 4220000000

    long nanosecs1 = start.until(end, ChronoUnit.NANOS); // 4220000000000
    long nanosecs2 = ChronoUnit.NANOS.between(start, end); // 4220000000000

    // Using others ChronoUnit will be thrown UnsupportedOperationException.
    // The following methods are examples thereof.
    long days1 = start.until(end, ChronoUnit.DAYS);
    long days2 = ChronoUnit.DAYS.between(start, end);
}
}

```

イントロ

`LocalTime`は、をすためにされるのクラスで、スレッドセーフであり、しばしばとなされます。はナノのでされます。たとえば、「1345.30.123456789」を`LocalTime`にできます。

このクラスは、またはタイムゾーンをまたはすことはありません。わりに、にされているのです。オフセットやタイムゾーンなどのなしでは、タイムラインでをすことはできません。これはベースのクラスで、`equals`メソッドをにするがあります。

フィールド

`MAX` - サポートされているローカルの、'235959.999999999'。ミッドナイト、ミン、ノーオン

メソッド

`now`、`nowClockClock`、`nowZoneId`ゾーン、`parseCharSequence text`

インスタンスメソッド

`isAfterLocalTime other`、`isBeforeLocalTime other`、マイナス`TemporalAmount amountToSubtract`、マイナス`long amountToSubtract`、`TemporalUnit`、プラス`TemporalAmount amountToAdd`、プラス`long amountToAdd`、`TemporalUnit`

```

ZoneId zone = ZoneId.of("Asia/Kolkata");
LocalTime now = LocalTime.now();
LocalTime now1 = LocalTime.now(zone);
LocalTime then = LocalTime.parse("04:16:40");

```

はのいずれかののでできます

```

long timeDiff = Duration.between(now, now1).toMinutes();
long timeDiff1 = java.time.temporal.ChronoUnit.MINUTES.between(now2, now1);

```

`LocalTime`のオブジェクトから、を/することもできます。

`minusNanoslong nanosToSubtract`、`minusSecondslong secondsToSubtract`、`plusHourslong hoursToSubtract`、`PlusMinuteslong hoursToMinutes`、`PlusNanoslong nanosToSubtract`、`PlusSecondslong secondsToSubtract`

```
now.plusHours(1L);  
now1.minusMinutes(20L);
```

オンラインでをむ <https://riptutorial.com/ja/java/topic/3065/>

177: なのためのAppDynamicsとTIBCO BusinessWorks

き

AppDynamicsは、アプリケーションのパフォーマンス、のスピードをするをすることとしているため、アプリケーションのデリバリデプロイメントは、DevOpsのりみをのにくためのなです。AppDをしたTIBCO BWアプリケーションのは、にはでがかかりませんが、なアプリケーションをするは、ながです。このガイドでは、にアプリケーションをすることなく、すべてのBWアプリケーションをにするをします。

Examples

AppdynamicsのすべてのBWアプリケーションをステップでする

1. TIBCO_HOME / bw / 5.12 / bin / bwengine.traLinuxのでTIBCO BW bwengine.traファイルをしてきます。
2. のようなをします。

。これらのみをしてください。

3. そのセクションののにのをします。tibco.deployment=tibco.deployment
4. ファイルのにし、にじてをのにきえるか、またはされないフラグをしてください
java.extended.properties = -javaagent/opt/appd/current/appagent/javaagent.jar -
Dappdynamics.http.proxyHost = -Dappdynamics.http.proxyPort = -
Dappdynamics.agent.applicationName = -Dappdynamics.agent.tierName = -
Dappdynamics.agent.nodeName =tibco.deployment-Dappdynamics.controller.ssl.enabled =
-Dappdynamics.controller.sslPort = -Dappdynamics.agent.logs.dir = -
Dappdynamics.agent.runtime.dir = -Dappdynamics.controller.hostName = -
Dappdynamics.controller.port = -Dappdynamics.agent.accountName = -
Dappdynamics.agent.accountAccessKey =
5. ファイルをしてデプロイします。デプロイに、すべてのアプリケーションにインストールするがあります。

オンラインでなのためのAppDynamicsとTIBCO BusinessWorksをむ

<https://riptutorial.com/ja/java/topic/10602/なのためのappdynamicsとtibco-businessworks>

178:

き

は、キーワード `extends` をして、あるクラスがのクラスのプロパティをし、するなオブジェクトです。インタフェースとキーワードの `implements` については、[インタフェース](#) をしてください。

- クラス `ClassB` は `ClassA` をする {...}
- `ClassB` クラスは `InterfaceA` {...} をしています
- `interfaceB` は `InterfaceA` をします {...}
- `ClassB` クラスが `ClassA` をして `InterfaceC`、`InterfaceD` {...} をする
- クラス `AbstractClassB` は `ClassA` をしています {...}
- クラス `AbstractClassB extends AbstractClassA` {...}
- クラス `AbstractClassB extends ClassA` は、`InterfaceC`、`InterfaceD` {...} をしています。

は、クラスが1つのパラメータをつように、ジェネリックとみわされることがよくあります。[ジェネリッククラスの](#) をしてください。

Examples

クラス

クラスは、`abstract` キーワードでマークされたクラスです。クラスとはに、なしのメソッドをむことができます。ただし、メソッドをたないクラスをすることはです。

クラスはインスタンスできません。サブクラスがクラスであるか、スーパークラスでクラスとしてマークされたすべてのメソッドをしているり、サブクラスすることができます。

クラスの

```
public abstract class Component {
    private int x, y;

    public setPosition(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public abstract void render();
}
```

なくとも1つのメソッドがある、クラスはクラスとマークするがあります。メソッドはされていないメソッドです。のサブクラスにのコードをするために、をつクラスでのメソッドをすることができます。

このクラスをインスタンスしようとすると、コンパイルエラーがします。

```
//error: Component is abstract; cannot be instantiated
Component myComponent = new Component();
```

しかし、`Component` をし、そのメソッドのすべてのをし、インスタンスすることができるクラス。

```
public class Button extends Component {

    @Override
    public void render() {
        //render a button
    }
}

public class TextBox extends Component {

    @Override
    public void render() {
        //render a textbox
    }
}
```

クラスをするインスタンスもクラスとしてキャストでき、メソッドがびされるとポリモーフィックなをしします。

```
Component myButton = new Button();
Component myTextBox = new TextBox();

myButton.render(); //renders a button
myTextBox.render(); //renders a text box
```

クラスとインタフェース

クラスとインタフェースのは、をするために/クラスをとしながらメソッドシングネチャをするをしします。

クラスとインタフェースのには2つのきないがあります。

- ・クラスはのクラスだけをすることができるが、くのインターフェースをすることができる。
- ・クラスにはインスタンス `static` フィールドをめることができますが、インタフェースには `static` フィールドしかめることはできません。

Java SE 8

インタフェースでされたメソッドにはがまれていないがあるため、メソッドとばれるのメソッドをするのにだったは、クラスがされました。

Java SE 8

Java 8では、インタフェースにデフォルトのメソッドをめることができます。これは、[インタ](#)

フェースのメソッドをしてされ、これでインタフェースとクラスをににします。

クラスのサブクラス

なJavaは、クラスのサブクラスのインスタンスのインスタンスをにするため、しいオブジェクトをするにメソッドのをします。のをすると、のようになります。

```
Component myAnonymousComponent = new Component() {
    @Override
    public void render() {
        // render a quick 1-time use component
    }
}
```

メソッドはのメソッドとにできますが、のメソッドとはなり、メソッドのオーバーライドをするためには「」メソッドをすることはです。スーパークラスのメソッドとじシグネチャをつメソッドをきむことは、オーバーライドのとわねますが、にはこれはにしいをしてのものをすだけです。

```
public class BaseClass {

    public static int num = 5;

    public static void sayHello() {
        System.out.println("Hello");
    }

    public static void main(String[] args) {
        BaseClass.sayHello();
        System.out.println("BaseClass's num: " + BaseClass.num);

        SubClass.sayHello();
        //This will be different than the above statement's output, since it runs
        //A different method
        SubClass.sayHello(true);

        StaticOverride.sayHello();
        System.out.println("StaticOverride's num: " + StaticOverride.num);
    }
}

public class SubClass extends BaseClass {

    //Inherits the sayHello function, but does not override it
    public static void sayHello(boolean test) {
        System.out.println("Hey");
    }
}

public static class StaticOverride extends BaseClass {

    //Hides the num field from BaseClass
    //You can even change the type, since this doesn't affect the signature
    public static String num = "test";

    //Cannot use @Override annotation, since this is static
}
```

```
//This overrides the sayHello method from BaseClass
public static void sayHello() {
    System.out.println("Static says Hi");
}

}
```

これらのクラスのいずれかをすると、のがされます。

```
Hello
BaseClass's num: 5
Hello
Hey
Static says Hi
StaticOverride's num: test
```

のとはなり、メソッドはされていないことにしてください。 `BaseClass.sayHello()` をって、いつでも `base sayHello` メソッドをびすことができます。しかし、サブクラスで `シグネチャ` をつメソッドがつからない、クラスはメソッドをします。2つのメソッドのシグネチャになる、たとえばじであっても、のメソッドをサブクラスからできます。

フィールドは、のでおいをします。

'final'をしてとオーバーライドをする

クラス

`class` でされるとき、`final` は、クラスを `extend` するのクラスのをぎます。 `final` クラスは、クラスの「リーフ」クラスです。

```
// This declares a final class
final class MyFinalClass {
    /* some code */
}

// Compilation error: cannot inherit from final MyFinalClass
class MySubClass extends MyFinalClass {
    /* more code */
}
```

クラスのユースケース

クラスを `private` コンストラクタとみわせて、クラスのインスタンスをまたはすることができます。メンバーのみをする、いわゆる "ユーティリティクラス" をするのにできます。すなわちとメソッド。

```
public final class UtilityClass {

    // Private constructor to replace the default visible constructor
```

```
private UtilityClass() {}

// Static members can still be used as usual
public static int doSomethingCool() {
    return 123;
}

}
```

なクラスも `final` としてするがあります。クラスとは、インスタンスがされたにできないものです。[.Immutableオブジェクト](#)のトピックをしてください。これにより、クラスのサブクラスをすることができなくなります。それは、サブタイプがそのスーパータイプの「」にうことをする [Liskov Substitution Principle](#) にします。

なからは、のクラスを `final` にすることで、プログラムのいをにすることができます。また、セキュリティサンドボックスでできないコードがされるシナリオでのセキュリティのします。たとえば、`String` は `final` としてされているため、できるクラスは、できないびしがにできるように、なサブクラスをけることにされることをするはありません。

`final` クラスの1つのは、Mockitoのようなフレームワークではしないことです。Mockitoバージョン2は、クラスのをサポートするようになりました。

なメソッド

`final` は、サブクラスでオーバーライドされないようにするメソッドにもできます。

```
public class MyClassWithFinalMethod {

    public final void someMethod() {
    }

}

public class MySubClass extends MyClassWithFinalMethod {

    @Override
    public void someMethod() { // Compiler error (overridden method is final)
    }

}
```

なメソッドは、、サブクラスをにすることなくクラスでできるものをするにされます。

`final` はにもできますが、の `final` のとはです。

リスコフ

Substitutabilityは、1987のでBarbara Liskovによってされたオブジェクトプログラミングのであり、クラス_Bがクラス_Aサブクラスであり、_Aがされるであれば_Bをわりにできるとべています。

```
class A {...}
```



```
class B extends A {...}

public void method(A obj) {...}

A a = new B(); // Assignment OK
method(new B()); // Passing as parameter OK
```

これは、タイプがインタフェースであり、オブジェクトにがないにもされます。

```
interface Foo {
    void bar();
}

class A implements Foo {
    void bar() {...}
}

class B implements Foo {
    void bar() {...}
}

List<Foo> foos = new ArrayList<>();
foos.add(new A()); // OK
foos.add(new B()); // OK
```

このリストには、じクラスにないオブジェクトがまれています。

クラスで `extends` キーワードをすると、スーパークラス クラスまたはクラス のすべてのプロパティがサブクラス クラスまたはクラス ともばれますにし、

```
public class BaseClass {

    public void baseMethod(){
        System.out.println("Doing base class stuff");
    }
}

public class SubClass extends BaseClass {

}
```

インスタンス `SubClass` メソッドのしている `baseMethod()`

```
SubClass s = new SubClass();
s.baseMethod(); //Valid, prints "Doing base class stuff"
```

コンテンツをサブクラスにすることができます。そのようにすると、クラスまたはそのじクラスのサブクラスにをえることなく、サブクラスにのがされます。

```
public class Subclass2 extends BaseClass {

    public void anotherMethod() {
        System.out.println("Doing subclass2 stuff");
    }
}
```

```

}

Subclass2 s2 = new Subclass2();
s2.baseMethod(); //Still valid , prints "Doing base class stuff"
s2.anotherMethod(); //Also valid, prints "Doing subclass2 stuff"

```

フィールドもされます

```

public class BaseClassWithField {

    public int x;

}

public class SubClassWithField extends BaseClassWithField {

    public SubClassWithField(int x) {
        this.x = x; //Can access fields
    }

}

```

`private` フィールドとメソッドはサブクラスにまだしますが、アクセスできません

```

public class BaseClassWithPrivateField {

    private int x = 5;

    public int getX() {
        return x;
    }

}

public class SubClassInheritsPrivateField extends BaseClassWithPrivateField {

    public void printX() {
        System.out.println(x); //Illegal, can't access private field x
        System.out.println(getX()); //Legal, prints 5
    }

}

SubClassInheritsPrivateField s = new SubClassInheritsPrivateField();
int x = s.getX(); //x will have a value of 5.

```

Javaでは、クラスはでも1つののクラスにできます。

```

public class A{}
public class B{}
public class ExtendsTwoClasses extends A, B {} //Illegal

```

これはとばれ、のではありませんが、Javaではクラスでそれをしません。

この、すべてのクラスには、すべてのクラスがする `Object` になるクラスのチェーンがあります。

とメソッド

Javaでは、クラスとクラスのがじのメソッドをつことができます。しかし、そのような、クラスのメソッドのはクラスのを**している**ので、メソッドオーバーライドではありません。えは

```
class StaticMethodTest {

    // static method and inheritance
    public static void main(String[] args) {
        Parent p = new Child();
        p.staticMethod(); // prints Inside Parent
        ((Child) p).staticMethod(); // prints Inside Child
    }

    static class Parent {
        public static void staticMethod() {
            System.out.println("Inside Parent");
        }
    }

    static class Child extends Parent {
        public static void staticMethod() {
            System.out.println("Inside Child");
        }
    }
}
```

メソッドはインスタンスにではなくクラスにバインドされ、このメソッドのバインドはコンパイラにわれます。 `staticMethod()` へののびしでクラスのリファレンス`p`がされたため、`Parent`のバージョンの`staticMethod()`がびされました。2のケースでは、`p`を`Child`クラスにキャストし、`Child`の`staticMethod()`しました。

シャドーイング

はSHADOWEDで、メソッドはOVERRIDDENです。どのがされるかは、そのがされているクラスにします。どのメソッドがされるかは、によってされるオブジェクトののクラスにします。

```
class Car {
    public int gearRatio = 8;

    public String accelerate() {
        return "Accelerate : Car";
    }
}

class SportsCar extends Car {
    public int gearRatio = 9;

    public String accelerate() {
        return "Accelerate : SportsCar";
    }

    public void test() {
    }
}
```

```

public static void main(String[] args) {

    Car car = new SportsCar();
    System.out.println(car.gearRatio + " " + car.accelerate());
    // will print out 8   Accelerate : SportsCar
}
}

```

オブジェクトのりみと

クラスのインスタンスをののようにサブクラスにキャストする $b = (B) a$; よりなクラスオブジェクトにクラスオブジェクトをりもうとしているのでナローイングとばれ、なキャストがです。

サブクラスのインスタンスをベースクラスにキャストィングするには $A a = b$; とばれ、キャストはありません。

として、のクラスとテストコードをえてみましょう。

```

class Vehicle {
}

class Car extends Vehicle {
}

class Truck extends Vehicle {
}

class Motorcycle extends Vehicle {
}

class Test {

    public static void main(String[] args) {

        Vehicle vehicle = new Car();
        Car car = new Car();

        vehicle = car; // is valid, no cast needed

        Car c = vehicle // not valid
        Car c = (Car) vehicle; //valid
    }
}

```

`Vehicle vehicle = new Car();`というステートメントがあり `Vehicle vehicle = new Car();`なJavaステートメントです。 `Car`すべてのインスタンスも `Vehicle`です。したがって、なキャストをとせずによりてがです。

、 `Car c = vehicle;`じゃない。な `vehicle`は `Vehicle`それはのインスタンスをすることを `Car`、トラック, オートバイ, or any other current or future subclass of. (Or indeed, an instance of itself, since we did not declare it as an class.) The assignment cannot be allowed, since that might lead to itself, since we did not declare it as an . (Or indeed, an instance of **Vehicle** itself, since we did not declare it as an . (Or indeed, an instance of class.) The assignment cannot be allowed, since that might lead to referring to a **Truck**のインスタンス referring to a です。

このようなをぐために、なキャストをするがあります。

```
Car c = (Car) vehicle;
```

キャストは、々はのをコンパイラえ`vehicle`あることを`Car`やサブクラスの`Car`。にじて、コンパイラはのチェックをするコードをします。チェックがすると、コードに`ClassCastException`がスローされます。

すべてのキャストがであるわけではないことにしてください。えば

```
String s = (String) vehicle; // not valid
```

Javaコンパイラは、`Vehicle`とのあるインスタンスは`String`とがないことをしています。キャストはしてすることはできませんでした。JLSはコンパイルエラーをきこすことをしています。

インタフェースへのプログラミング

インターフェイスへのプログラミングのにあるえは、コードをにインターフェイスについており、インスタンスにはコンクリートクラスのみをすることです。このコンテキストでは、Javaコレクションをういコードは、のようになりますメソッドはまったくされているわけではありません。

```
public <T> Set<T> toSet(Collection<T> collection) {  
    return Sets.newHashSet(collection);  
}
```

いコードはこのようにえるかもしれません

```
public <T> HashSet<T> toSet(ArrayList<T> collection) {  
    return Sets.newHashSet(collection);  
}
```

はなののでできるだけでなく、そのはのがするコードとよりがあり、にはインターフェイスへのプログラミングのコンセプトにいます。ただし、をするもなはのとおりです。

- ほとんどの、がされるコンテキストは、ながするくのをとせず、としないはずです。
- インタフェースをすることで、よりされたコードがされ、いくつかののシナリオにするクラスにさらにのパブリックメソッドがされます。
- インタフェースはにであるため、コードはよりテストです。
- に、このコンセプトは、なくともテストのために1つののみがされるでもちます。

したがって、あるのをにいてしいコードをくときに、どのようにしてプログラミングのをインターフェイスににできますかにする1つのオプションは、のパターンのみわせです。

- インターフェイスへのプログラミング
-

- ビルダー

これらのにつくのは、いくつかのなるプロトコルにされたRPCのされたりバージョンです。

```
public interface RemoteInvoker {  
    <RQ, RS> CompletableFuture<RS> invoke(RQ request, Class<RS> responseClass);  
}
```

のインタフェースは、ファクトリをしてインスタンスされることはされていません。わりに、HTTPびしとAMQPの1つずつのなインタフェースをします。それぞれは、ファクトリとインスタンスをするビルダをちます。のインターフェース

```
public interface AmqpInvoker extends RemoteInvoker {  
    static AmqpInvokerBuilder with(String instanceId, ConnectionFactory factory) {  
        return new AmqpInvokerBuilder(instanceId, factory);  
    }  
}
```

AMQPですのためのRemoteInvokerインスタンスは、ビルダーにじてにまたはにみてることことができるようになりました。

```
RemoteInvoker invoker = AmqpInvoker.with(instanceId, factory)  
    .requestRouter(router)  
    .build();
```

のびしはのようになります。

```
Response res = invoker.invoke(new Request(data), Response.class).get();
```

Java 8では、メソッドをインターフェイスにできるため、AmqpInvoker.with() きえられたのコードではなファクトリがになりました。バージョン8よりのJavaでは、のFactoryクラスでもじがられます。

```
public interface AmqpInvoker extends RemoteInvoker {  
    class Factory {  
        public static AmqpInvokerBuilder with(String instanceId, ConnectionFactory factory) {  
            return new AmqpInvokerBuilder(instanceId, factory);  
        }  
    }  
}
```

するインスタンスは、のようになります。

```
RemoteInvoker invoker = AmqpInvoker.Factory.with(instanceId, factory)  
    .requestRouter(router)  
    .build();
```

のビルダーはこのようにえるかもしれませんがのものはデフォルトからした15までのパラメーターをできるのでしてはいますがこのはパブリックではないので、のAmqpInvokerインタフェースから

のみできます。

```
public class AmqpInvokerBuilder {
    ...
    AmqpInvokerBuilder(String instanceId, ConnectionFactory factory) {
        this.instanceId = instanceId;
        this.factory = factory;
    }

    public AmqpInvokerBuilder requestRouter(RequestRouter requestRouter) {
        this.requestRouter = requestRouter;
        return this;
    }

    public AmqpInvoker build() throws TimeoutException, IOException {
        return new AmqpInvokerImpl(instanceId, factory, requestRouter);
    }
}
```

に、ビルダーはFreeBuilderのようなツールを使ってすることもできます。

に、このインタフェースのなぞしてのされるは、インタフェース、ファクトリ、およびビルダーのをするパッケージローカルクラスとしてされています。

```
class AmqpInvokerImpl implements AmqpInvoker {
    AmqpInvokerImpl(String instanceId, ConnectionFactory factory, RequestRouter requestRouter) {
        ...
    }

    @Override
    public <RQ, RS> CompletableFuture<RS> invoke(final RQ request, final Class<RS> respClass) {
        ...
    }
}
```

、このパターンは、がどれほどかにかかわらず、しいコードをすべてするのににであることがわかりました。

クラスとインタフェースの **"Is-a"** リレーションと **"Has-a"** リレーション

クラスををするのオブジェクトでじまたはなるをする

インタフェースををするのなオブジェクトによるをする

クラスは、は「」をし、インターフェースは「つ」をします。

これはのコードでることができます

```
public class InterfaceAndAbstractClassDemo{
    public static void main(String args[]){

        Dog dog = new Dog("Jack",16);
        Cat cat = new Cat("Joe",20);
    }
}
```

```

        System.out.println("Dog:"+dog);
        System.out.println("Cat:"+cat);

        dog.remember();
        dog.protectOwner();
        Learn dl = dog;
        dl.learn();

        cat.remember();
        cat.protectOwner();

        Climb c = cat;
        c.climb();

        Man man = new Man("Ravindra",40);
        System.out.println(man);

        Climb cm = man;
        cm.climb();
        Think t = man;
        t.think();
        Learn l = man;
        l.learn();
        Apply a = man;
        a.apply();
    }
}

abstract class Animal{
    String name;
    int lifeExpentency;
    public Animal(String name,int lifeExpentency ){
        this.name = name;
        this.lifeExpentency=lifeExpentency;
    }
    public abstract void remember();
    public abstract void protectOwner();

    public String toString(){
        return this.getClass().getSimpleName()+":"+name+": "+lifeExpentency;
    }
}

class Dog extends Animal implements Learn{

    public Dog(String name,int age){
        super(name,age);
    }
    public void remember(){
        System.out.println(this.getClass().getSimpleName()+" can remember for 5 minutes");
    }
    public void protectOwner(){
        System.out.println(this.getClass().getSimpleName()+ " will protect owner");
    }
    public void learn(){
        System.out.println(this.getClass().getSimpleName()+ " can learn:");
    }
}

class Cat extends Animal implements Climb {
    public Cat(String name,int age){
        super(name,age);
    }
}

```



```

    public void remember(){
        System.out.println(this.getClass().getSimpleName() + " can remember for 16 hours");
    }
    public void protectOwner(){
        System.out.println(this.getClass().getSimpleName() + " won't protect owner");
    }
    public void climb(){
        System.out.println(this.getClass().getSimpleName() + " can climb");
    }
}
interface Climb{
    void climb();
}
interface Think {
    void think();
}
interface Learn {
    void learn();
}
interface Apply{
    void apply();
}

class Man implements Think,Learn,Apply,Climb{
    String name;
    int age;

    public Man(String name,int age){
        this.name = name;
        this.age = age;
    }
    public void think(){
        System.out.println("I can think:"+this.getClass().getSimpleName());
    }
    public void learn(){
        System.out.println("I can learn:"+this.getClass().getSimpleName());
    }
    public void apply(){
        System.out.println("I can apply:"+this.getClass().getSimpleName());
    }
    public void climb(){
        System.out.println("I can climb:"+this.getClass().getSimpleName());
    }
    public String toString(){
        return "Man :"+name+":Age:"+age;
    }
}

```

```

Dog:Dog:Jack:16
Cat:Cat:Joe:20
Dog can remember for 5 minutes
Dog will protect owner
Dog can learn:
Cat can remember for 16 hours
Cat won't protect owner
Cat can climb
Man :Ravindra:Age:40
I can climb:Man
I can think:Man

```

```
I can learn:Man
I can apply:Man
```

キーノート

1. `Animal`は、をつクラスです `name`と`lifeExpectancy`およびabstractメソッド `remember()`と`protectOwner()`。 `Dog`と`Cat`は、 `remember()` `protectOwner()`と`protectOwner()`メソッドをした `Animals`です。
2. `Cat`は`climb()`ことができますが、`Dog`はできません。 `Dog`は`think()`ことができ`think()`が、 `Cat`はできません。これらののは、によって`Cat`および`Dog`されます。
3. `Man`は`Animal`はありませんが、 `Think`、 `Learn`、 `Apply`、 `Climb`ことができます。
4. `Cat`は`Man`はありませんが、 `Climb`ことができます。
5. `Dog`は`Man`はありませんが、それは`Learn`ことができます
6. `Man`は`Cat`も`Dog`もないが、 `Animal`、 `Cat`、または`Dog`をばすことなくのいくつかをつことができる。これはインタフェースでわれます。
7. `Animal`はクラスであるにもかかわらず、インターフェイスとはなり、コンストラクタをちます。

TL; DR

のクラスはインタフェースをしてをつことができますが、するクラスはクラスのによってをします。

のユースケースでどのドキュメントをするかをするには、Javaのマニュアル[ページ](#)をしてください。

のはクラスのをしてください

1. いくつかのにするクラスのでコードをしたいとします。
2. クラスをするクラスには、くの名メソッドやフィールドがあり、`public`のアクセス`protected`や`private`などがです。
3. フィールドまたはフィールドをしたいとします。

のはインタフェースのをしてください

1. のクラスがあなたのインターフェイスをすることをします。たとえば、なくのオブジェクトが`Serializable`インターフェイスをできます。
2. のデータのるいをしたいが、そのるいをがするかはしていない。
3. あなたはのをしたいとっています。

でオーバーライドする

のオーバーライドは、サブクラスのスーパークラスからすでにされたメソッドをするにされますが、スーパークラスで々されたとはなりません。オーバーライドをすると、のをしてコードをし、ユーザーのニーズにわせてすることができます。

のは、`ClassB`がメソッドをじてされるをすることによって`ClassA`のをオーバーライドするをしてい`ClassB`。

```
public static void main(String[] args) {
    ClassA a = new ClassA();
    ClassA b = new ClassB();
    a.printing();
    b.printing();
}

class ClassA {
    public void printing() {
        System.out.println("A");
    }
}

class ClassB extends ClassA {
    public void printing() {
        System.out.println("B");
    }
}
```

A

B

オンラインでをむ <https://riptutorial.com/ja/java/topic/87/>

179: リリースのJARファイル

き

Java 9でされたの1つは、じJarファイルののJavaリリースをターゲットとするコードをバンドルできるマルチリリースJarMRJARです。このは、[JEP 238](#)でされています。

Examples

マルチリリースJarファイルのの

MANIFEST.MFファイルで`Multi-Release: true`すると、JarファイルはマルチリリースJarになり、JavaランタイムはMRJARをサポートしているりのメジャーバージョンにじてなバージョンのクラスをします。

このようなJarのはのとおりです。

```
jar root
- A.class
- B.class
- C.class
- D.class
- META-INF
  - versions
    - 9
      - A.class
      - B.class
    - 10
      - A.class
```

- JDK <9では、ルートエントリのクラスのみがJavaランタイムにされます。
- JDK 9では、クラスAとBは`root/META-INF/versions/9`ディレクトリ`root/META-INF/versions/9`からロードされ、CとDはベースエントリからロードされます。
- JDK 10では、クラスAは`root/META-INF/versions/10`ディレクトリからロードされます。

jarツールをしてマルチリリースJarをする

jar コマンドをすると、Java 8とJava 9のでコンパイルされたじクラスの2つのバージョンをむマルチリリースJarをできますが、クラスがじであることをすがされます。

```
C:\Users\manouti>jar --create --file MR.jar -C sampleproject-base demo --release 9 -C
sampleproject-9 demo
Warning: entry META-INF/versions/9/demo/SampleClass.class contains a class that
is identical to an entry already in the jar
```

--release 9 オプションは、MRJARのバージョンされたエントリ、つまり`root/META-INF/versions/9`

に `jar` にくすべてのもの `sampleproject-9` ディレクトリの `demo` パッケージをめるようにします。のはです。

```
jar root
- demo
  - SampleClass.class
- META-INF
  - versions
    - 9
      - demo
        - SampleClass.class
```

`SampleClass` のURLをし、Java 9バージョンにする`Main`というクラスをしましょう。

```
package demo;

import java.net.URL;

public class Main {

    public static void main(String[] args) throws Exception {
        URL url = Main.class.getClassLoader().getResource("demo/SampleClass.class");
        System.out.println(url);
    }
}
```

このクラスをコンパイルして`jar`コマンドをすると、エラーがします。

```
C:\Users\manouti>jar --create --file MR.jar -C sampleproject-base demo --release 9 -C
sampleproject-9 demoentry: META-INF/versions/9/demo/Main.class, contains a new public class
not found in base entries
Warning: entry META-INF/versions/9/demo/Main.java, multiple resources with same name
Warning: entry META-INF/versions/9/demo/SampleClass.class contains a class that
is identical to an entry already in the jar
invalid multi-release jar file MR.jar deleted
```

そのは、`jar` ツールは、ベースエントリにもされていないでも、パブリッククラスをバージョンされたエントリにすることをぐためです。これは、MRJARがなるJavaバージョンのじAPIをするようにわれま。には、このルールはではありません。それは、`jar` ようなツールによってのみされるかもしれま。こののケースでは、`Main` のはサンプルコードをすることで、エントリにコピーをするだけです。クラスがJava 9になしいのであれば、にすることができます。

`Main` をルートエントリにするには、にJava 9よりのリリースをにコンパイルするがあります。これは、`javac` しい`--release` オプションをしてうことができます。

```
C:\Users\manouti\sampleproject-base\demo>javac --release 8 Main.java
C:\Users\manouti\sampleproject-base\demo>cd ../../
C:\Users\manouti>jar --create --file MR.jar -C sampleproject-base demo --release 9 -C
sampleproject-9 demo
```

`Main` クラスをすると、`SampleClass` がバージョンされたディレクトリからロードされます。

```
C:\Users\manouti>java --class-path MR.jar demo.Main
jar:file:/C:/Users/manouti/MR.jar!/META-INF/versions/9/demo/SampleClass.class
```

マルチリリースJarのロードされたクラスのURL

のマルチリリースのJarがあるとします。

```
jar root
- demo
  - SampleClass.class
- META-INF
  - versions
    - 9
      - demo
        - SampleClass.class
```

のクラスは、SampleClassのURLをします。

```
package demo;

import java.net.URL;

public class Main {

    public static void main(String[] args) throws Exception {
        URL url = Main.class.getClassLoader().getResource("demo/SampleClass.class");
        System.out.println(url);
    }
}
```

クラスがコンパイルされ、Java 9のバージョンされたエントリにMRJARでされた、するとようになります。

```
C:\Users\manouti>java --class-path MR.jar demo.Main
jar:file:/C:/Users/manouti/MR.jar!/META-INF/versions/9/demo/SampleClass.class
```

オンラインでリリースのJARファイルをむ <https://riptutorial.com/ja/java/topic/9866/> リリースのjar
ファイル

Examples

イベントリスナーの

`Preferences` オブジェクトによって `NodeChangeEvent` されるイベントには、`PreferenceChangeEvent` と `NodeChangeEvent` 2があります。

PreferenceChangeEvent

ノードのキーとのペアのいずれかがされるたびに、`Properties` オブジェクトによって `PreferenceChangeEvent` がされ `Properties`。 `PreferenceChangeEvent` は、`PreferenceChangeListener` ことができます

Java SE 8

```
preferences.addPreferenceChangeListener(evt -> {
    String newValue = evt.getNewValue();
    String changedPreferenceKey = evt.getKey();
    Preferences changedNode = evt.getNode();
});
```

Java SE 8

```
preferences.addPreferenceChangeListener(new PreferenceChangeListener() {
    @Override
    public void preferenceChange(PreferenceChangeEvent evt) {
        String newValue = evt.getNewValue();
        String changedPreferenceKey = evt.getKey();
        Preferences changedNode = evt.getNode();
    }
});
```

このリスナーは、されたノードのキーとのペアをリッスンしません。

NodeChangeEvent

このイベントは、`Properties` ノードのノードがまたはされるたびに `Properties`。

```
preferences.addNodeChangeListener(new NodeChangeListener() {
    @Override
    public void childAdded(NodeChangeEvent evt) {
        Preferences addedChild = evt.getChild();
        Preferences parentOfAddedChild = evt.getParent();
    }

    @Override
    public void childRemoved(NodeChangeEvent evt) {
        Preferences removedChild = evt.getChild();
    }
});
```

```

        Preferences parentOfRemovedChild = evt.getParent();
    }
});

```

のサブノードをする

Preferences オブジェクトは、に Preferences ツリーのののノードをします。

```

/userRoot
├── com
│   └── mycompany
│       └── myapp
│           ├── darkApplicationMode=true
│           ├── showExitConfirmation=false
│           └── windowMaximized=true
└── org
    └── myorganization
        ├── anotherapp
        │   ├── defaultFont=Helvetica
        │   ├── defaultSavePath=/home/matt/Documents
        │   └── exporting
        │       ├── defaultFormat=pdf
        │       └── openInBrowserAfterExport=false

```

/com/mycompany/myapp ノードを返すには

1. により、クラスのパッケージにづいて

```

package com.mycompany.myapp;

// ...

// Because this class is in the com.mycompany.myapp package, the node
// /com/mycompany/myapp will be returned.
Preferences myApp = Preferences.userNodeForPackage(getClass());

```

2. パス

```

Preferences myApp = Preferences.userRoot().node("com/mycompany/myapp");

```

パス / まらないパスをすると、されたノードをにしてパスがされます。たとえば、のでは、パス /one/two/three/com/mycompany/myapp のノードがされます。

```

Preferences prefix = Preferences.userRoot().node("one/two/three");
Preferences myAppWithPrefix = prefix.node("com/mycompany/myapp");
// prefix is /one/two/three
// myAppWithPrefix is /one/two/three/com/mycompany/myapp

```

3. パスによる

```

Preferences myApp = Preferences.userRoot().node("/com/mycompany/myapp");

```

ルートノードでパスをすることは、パスをすることと異なることはありません。は、サブノードでびされた、ルートノードににしてパスがされることです。

```

Preferences prefix = Preferences.userRoot().node("one/two/three");
Preferences myAppWithoutPrefix = prefix.node("/com/mycompany/myapp");

```



```
// prefix           is /one/two/three
// myAppWithoutPrefix is /com/mycompany/myapp
```

このアプリケーションインスタンスにわたるこの

Preferencesすべてのインスタンスは、このJavaマシンのJVMのスレッドでスレッドセーフです。PreferencesはこのJVMで実行できるため、このマシンで実行させる必要があります。

このインスタンスでのみ実行されるはずのアプリケーションがある、このはありません。

このシステムにこのインスタンスで実行され、したがって、アプリケーションでこのPreferencesアクセスは、システムのJVMで実行する必要があるが、その、**sync()**のこのPreferencesノードは、このを実行するために実行することができるPreferencesノードでありますシステムのこのJVMから返る

```
// Warning: don't use this if your application is intended
// to only run a single instance on a machine once
// (this is probably the case for most desktop applications)
try {
    preferences.sync();
} catch (BackingStoreException e) {
    // Deal with any errors while saving the preferences to the backing storage
    e.printStackTrace();
}
```

こののエクスポート

Preferencesノードは、そのノードをXMLにエクスポートできます。このXMLツリーをインポートできます。このXMLは、ユーザーまたはシステムの「Preferencesからエクスポートされたかどうかを返しています。

このノードをエクスポートするが、そのノードはエクスポートしない

Java SE 7

```
try (OutputStream os = ...) {
    preferences.exportNode(os);
} catch (IOException ioe) {
    // Exception whilst writing data to the OutputStream
    ioe.printStackTrace();
} catch (BackingStoreException bse) {
    // Exception whilst reading from the backing preferences store
    bse.printStackTrace();
}
```

Java SE 7

```
OutputStream os = null;
try {
    os = ...;
    preferences.exportSubtree(os);
} catch (IOException ioe) {
    // Exception whilst writing data to the OutputStream
    ioe.printStackTrace();
} catch (BackingStoreException bse) {
    // Exception whilst reading from the backing preferences store
    bse.printStackTrace();
} finally {
    if (os != null) {
        try {
```

```

        os.close();
    } catch (IOException ignored) {}
}
}

```

ノードをのノードをエクスポートするには

Java SE 7

```

try (OutputStream os = ...) {
    preferences.exportNode(os);
} catch (IOException ioe) {
    // Exception whilst writing data to the OutputStream
    ioe.printStackTrace();
} catch (BackingStoreException bse) {
    // Exception whilst reading from the backing preferences store
    bse.printStackTrace();
}

```

Java SE 7

```

OutputStream os = null;
try {
    os = ...;
    preferences.exportSubtree(os);
} catch (IOException ioe) {
    // Exception whilst writing data to the OutputStream
    ioe.printStackTrace();
} catch (BackingStoreException bse) {
    // Exception whilst reading from the backing preferences store
    bse.printStackTrace();
} finally {
    if (os != null) {
        try {
            os.close();
        } catch (IOException ignored) {}
    }
}

```

のインポート

PreferencesノードはXMLからインポートできます。インポートは、なするXMLドキュメントをするため、Preferencesのエクスポートとすることをとしています。

XMLは、それらがユーザーまたはシステムの「Preferencesからエクスポートされたかどうかをえています。したがって、らはどこからたのかすることなく、それぞれのPreferencesツリーにインポートすることができます。は、XMLがユーザーまたはシステムPreferencesからエクスポートされたかどうかをにし、エクスポートされたツリーににインポートします。

Java SE 7

```

try (InputStream is = ...) {
    // This is a static call on the Preferences class
    Preferences.importPreferences(is);
} catch (IOException ioe) {
    // Exception whilst reading data from the InputStream
    ioe.printStackTrace();
} catch (InvalidPreferencesFormatException ipfe) {
    // Exception whilst parsing the XML document tree
}

```

```

        ipfe.printStackTrace();
    }

```

Java SE 7

```

InputStream is = null;
try {
    is = ...;
    // This is a static call on the Preferences class
    Preferences.importPreferences(is);
} catch (IOException ioe) {
    // Exception whilst reading data from the InputStream
    ioe.printStackTrace();
} catch (InvalidPreferencesFormatException ipfe) {
    // Exception whilst parsing the XML document tree
    ipfe.printStackTrace();
} finally {
    if (is != null) {
        try {
            is.close();
        } catch (IOException ignored) {}
    }
}

```

イベントリスナーの

イベントリスナーは、のPropertiesノードからできますが、リスナーのインスタンスを する があります。

Java SE 8

```

Preferences preferences = Preferences.userNodeForPackage(getClass());

PreferenceChangeListener listener = evt -> {
    System.out.println(evt.getKey() + " got new value " + evt.getNewValue());
};
preferences.addPreferenceChangeListener(listener);

//
// later...
//

preferences.removePreferenceChangeListener(listener);

```

Java SE 8

```

Preferences preferences = Preferences.userNodeForPackage(getClass());

PreferenceChangeListener listener = new PreferenceChangeListener() {
    @Override
    public void preferenceChange(PreferenceChangeEvent evt) {
        System.out.println(evt.getKey() + " got new value " + evt.getNewValue());
    }
};
preferences.addPreferenceChangeListener(listener);

//
// later...
//

```

```
preferences.removePreferenceChangeListener(listener);
```

NodeChangeListenerもです。

を

Preferencesノードのは、String、boolean、byte[]、double、float、intまたはlongいずれかにlongます。されたがPreferencesノードにしないは、すべてのびしでデフォルトを

```
Preferences preferences = Preferences.userNodeForPackage(getClass());

String someString = preferences.get("someKey", "this is the default value");
boolean someBoolean = preferences.getBoolean("someKey", true);
byte[] someByteArray = preferences.getByteArray("someKey", new byte[0]);
double someDouble = preferences.getDouble("someKey", 887284.4d);
float someFloat = preferences.getFloat("someKey", 38723.3f);
int someInt = preferences.getInt("someKey", 13232);
long someLong = preferences.getLong("someKey", 2827637868234L);
```

の

Preferencesノードにをするには、putXXX()メソッドの1つがされます。Preferencesノードのは、String、boolean、byte[]、double、float、intまたはlongいずれかにlongます。

```
Preferences preferences = Preferences.userNodeForPackage(getClass());

preferences.put("someKey", "some String value");
preferences.putBoolean("someKey", false);
preferences.putByteArray("someKey", new byte[0]);
preferences.putDouble("someKey", 187398123.4454d);
preferences.putFloat("someKey", 298321.445f);
preferences.putInt("someKey", 77637);
preferences.putLong("someKey", 2873984729834L);
```

の

Preferencesを、ユーザーのなアプリケーション、例えばエディタフォント、アプリケーションをフルスクリーンモードでするかどうか、「これをもうしない」チェックボックス、そうですね。

```
public class ExitConfirmer {
    private static boolean confirmExit() {
        Preferences preferences = Preferences.userNodeForPackage(ExitConfirmer.class);
        boolean doShowDialog = preferences.getBoolean("showExitConfirmation", true); // true
        is default value

        if (!doShowDialog) {
            return true;
        }

        //
        // Show a dialog here...
        //
        boolean exitWasConfirmed = ...; // whether the user clicked OK or Cancel
        boolean doNotShowAgain = ...; // get value from "Do not show again" checkbox

        if (exitWasConfirmed && doNotShowAgain) {
```

```
        // Exit was confirmed and the user chose that the dialog should not be shown again
        // Save these settings to the Preferences object so the dialog will not show again
next time
        preferences.putBoolean("showExitConfirmation", false);
    }

    return exitWasConfirmed;
}

public static void exit() {
    if (confirmExit()) {
        System.exit(0);
    }
}
}
```

オンラインでコードを編集 <https://riptutorial.com/ja/java/topic/582>

181: と

き

リーダとライタとそれらのそれぞれのサブクラスは、テキストベースのデータに する な / を します。

Examples

BufferedReader

き

BufferedReaderクラスは、に の2つの に される のReaderクラスのラッパーです。

1. BufferedReaderは、ラップされたReaderバッファリングを します。これにより、アプリケーションは の / オオーバーヘッドを させることなく、 に1つの を み ることができます。
2. BufferedReaderは、 に1のテキストを み む を します。

BufferedReaderを するための

BufferedReaderを するための のパターンは のとおりです。まず、 を みたいReaderを します。 に、 ReaderをラップするBufferedReaderをインスタンス します。 に、 データを み みます。 に、ラップされた `Readerを じるBufferedReaderを します。 えば

```
File someFile = new File(...);
int aCount = 0;
try (FileReader fr = new FileReader(someFile);
    BufferedReader br = new BufferedReader(fr)) {
    // Count the number of 'a' characters.
    int ch;
    while ((ch = br.read()) != -1) {
        if (ch == 'a') {
            aCount++;
        }
    }
    System.out.println("There are " + aCount + " 'a' characters in " + someFile);
}
```

このパターンは のReader できます

ノート

1. Java 7 またはそれ の `try-with-resources` を して、 となるリーダーが に じられていることを しました。これにより、 なるリソースリークが されます。 のバージョンのJavaでは、 `finally` ブロック の `BufferedReader` を に じます。
2. `try` ブロック のコードは、 `FileReader` から み った に するコードと じです。 、 `BufferedReader` は、ラップするReaderとまったく じように します。 いは、 このバージョンは ずっと だということです。

BufferedReader バッファサイズ

BufferedReader.readLine() メソッド

ファイルのすべての をリストに み む

これは、ファイルの各行を1行ずつ読み取り、List<String>に格納することによって行われます。リストが返されます。

```
public List<String> getAllLines(String filename) throws IOException {
    List<String> lines = new ArrayList<String>();
    try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
        String line = null;
        while ((line = reader.readLine()) != null) {
            lines.add(line);
        }
    }
    return lines;
}
```

Java 8では、lines()メソッドを利用してこれを簡潔にするためのより簡潔なコードが提供されています。

```
public List<String> getAllLines(String filename) throws IOException {
    try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
        return br.lines().collect(Collectors.toList());
    }
    return Collections.empty();
}
```

StringWriterの概要

Java StringWriterクラスは、データを格納するために利用できるバッファからのデータを格納するストリームです。

StringWriterクラスは、Writerクラスを継承します。

StringWriterクラスでは、ネットワークソケットやファイルなどのシステムリソースは使われないため、StringWriterを利用する際にはありません。

```
import java.io.*;

public class StringWriterDemo {
    public static void main(String[] args) throws IOException {
        char[] ary = new char[1024];
        StringWriter writer = new StringWriter();
        FileInputStream input = null;
        BufferedReader buffer = null;
        input = new FileInputStream("c://stringwriter.txt");
        buffer = new BufferedReader(new InputStreamReader(input, "UTF-8"));
        int x;
        while ((x = buffer.read(ary)) != -1) {
            writer.write(ary, 0, x);
        }
        System.out.println(writer.toString());
        writer.close();
        buffer.close();
    }
}
```

このコードは、BufferedReaderを利用してストリームからファイルデータを読み取り、StringWriterのwrite()メソッドを利用して書き出すのに使われます。

オンラインでコードを動かす <https://riptutorial.com/ja/java/topic/10618> と参照してください。

□□

- `javap [options] <classes>`

パラメーター

<classes>	アセンブルするクラスのリスト。 <code>package1.package2.Classname</code> 、または <code>package1/package2/Classname</code> のいずれかに <code>package1.package2.Classname</code> ます。 。 <code>.class</code> はめないでください。
<code>-help</code> 、 <code>-? --help</code> 、 <code>-?</code>	このメッセージをする
<code>-version</code>	バージョン
<code>-v</code> 、 <code>-verbose</code>	をする
<code>-l</code>	とローカルテーブルをする
<code>-public</code>	パブリッククラスとメンバーのみをする
<code>-protected</code>	された/クラスとメンバーをする
<code>-package</code>	パッケージ/された/されているクラスとメンバーをするデフォルト
<code>-p</code> 、 <code>-private</code>	すべてのクラスとメンバーをする
<code>-c</code>	コードをアセンブルする
<code>-s</code>	をする
<code>-sysinfo</code>	のクラスのシステムパス、サイズ、、MD5ハッシュをする
<code>-constants</code>	をする
<code>-classpath <path></code>	ユーザークラスファイルのを
<code>-cp <path></code>	ユーザークラスファイルのを
<code>-bootclasspath <path></code>	ブートストラップクラスファイルのをきする

Examples

`javap` でバイトコードを□□する

☐☐されたバイトコードをJavaプログラムに☐たい☐☐は、☐☐されたjavapコマンドを☐☐してそれを☐☐できます。

☐たちが☐のJavaソースファイルを☐っている☐と☐☐します☐

```
package com.stackoverflow.documentation;

import org.springframework.stereotype.Service;

import java.io.IOException;
import java.io.InputStream;
import java.util.List;

@Service
public class HelloWorldService {

    public void sayHello() {
        System.out.println("Hello, World!");
    }

    private Object[] pvtMethod(List<String> strings) {
        return new Object[]{strings};
    }

    protected String tryCatchResources(String filename) throws IOException {
        try (InputStream inputStream = getClass().getResourceAsStream(filename)) {
            byte[] bytes = new byte[8192];
            int read = inputStream.read(bytes);
            return new String(bytes, 0, read);
        } catch (IOException | RuntimeException e) {
            e.printStackTrace();
            throw e;
        }
    }

    void stuff() {
        System.out.println("stuff");
    }
}
```

ソースファイルをコンパイルした☐、☐も☐☐ない☐は☐のとおりです。

```
cd <directory containing classes> (e.g. target/classes)
javap com/stackoverflow/documentation/SpringExample
```

☐☐を☐☐する

```
Compiled from "HelloWorldService.java"
public class com.stackoverflow.documentation.HelloWorldService {
    public com.stackoverflow.documentation.HelloWorldService();
    public void sayHello();
    protected java.lang.String tryCatchResources(java.lang.String) throws java.io.IOException;
    void stuff();
}
```

これは、クラス☐のすべての☐プライベートメソッドをリストしますが、ほとんどの☐☐にはそれほど☐に☐ちません。☐☐のコマンドはもっと☐☐です☐

```
javap -p -c -s -constants -l -v com/stackoverflow/documentation/HelloWorldService
```

☐☐を☐☐する☐

```

Classfile /Users/pivotal/IdeaProjects/stackoverflow-spring-
docs/target/classes/com/stackoverflow/documentation/HelloWorldService.class
  Last modified Jul 22, 2016; size 2167 bytes
  MD5 checksum 6e33b5c292ead21701906353b7f06330
  Compiled from "HelloWorldService.java"
public class com.stackoverflow.documentation.HelloWorldService
  minor version: 0
  major version: 51
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
  #1 = Methodref          #5.#60      // java/lang/Object."<init>":()V
  #2 = Fieldref           #61.#62      // java/lang/System.out:Ljava/io/PrintStream;
  #3 = String              #63          // Hello, World!
  #4 = Methodref          #64.#65      // java/io/PrintStream.println:(Ljava/lang/String;)V
  #5 = Class                #66          // java/lang/Object
  #6 = Methodref          #5.#67      // java/lang/Object.getClass:()Ljava/lang/Class;
  #7 = Methodref          #68.#69      //
java/lang/Class.getResourceAsStream:(Ljava/lang/String;)Ljava/io/InputStream;
  #8 = Methodref          #70.#71      // java/io/InputStream.read:([B)I
  #9 = Class                #72          // java/lang/String
 #10 = Methodref          #9.#73       // java/lang/String."<init>":([BII)V
 #11 = Methodref          #70.#74      // java/io/InputStream.close:()V
 #12 = Class                #75          // java/lang/Throwable
 #13 = Methodref          #12.#76      //
java/lang/Throwable.addSuppressed:(Ljava/lang/Throwable;)V
 #14 = Class                #77          // java/io/IOException
 #15 = Class                #78          // java/lang/RuntimeException
 #16 = Methodref          #79.#80      // java/lang/Exception.printStackTrace:()V
 #17 = String              #55          // stuff
 #18 = Class                #81          // com/stackoverflow/documentation/HelloWorldService
 #19 = Utf8                <init>
 #20 = Utf8                ()V
 #21 = Utf8                Code
 #22 = Utf8                LineNumberTable
 #23 = Utf8                LocalVariableTable
 #24 = Utf8                this
 #25 = Utf8                Lcom/stackoverflow/documentation/HelloWorldService;
 #26 = Utf8                sayHello
 #27 = Utf8                pvtMethod
 #28 = Utf8                (Ljava/util/List;) [Ljava/lang/Object;
 #29 = Utf8                strings
 #30 = Utf8                Ljava/util/List;
 #31 = Utf8                LocalVariableTypeTable
 #32 = Utf8                Ljava/util/List<Ljava/lang/String;>;
 #33 = Utf8                Signature
 #34 = Utf8                (Ljava/util/List<Ljava/lang/String;>;) [Ljava/lang/Object;
 #35 = Utf8                tryCatchResources
 #36 = Utf8                (Ljava/lang/String;)Ljava/lang/String;
 #37 = Utf8                bytes
 #38 = Utf8                [B
 #39 = Utf8                read
 #40 = Utf8                I
 #41 = Utf8                inputStream
 #42 = Utf8                Ljava/io/InputStream;
 #43 = Utf8                e
 #44 = Utf8                Ljava/lang/Exception;
 #45 = Utf8                filename
 #46 = Utf8                Ljava/lang/String;
 #47 = Utf8                StackMapTable
 #48 = Class                #81          // com/stackoverflow/documentation/HelloWorldService
 #49 = Class                #72          // java/lang/String

```

```

#50 = Class          #82          // java/io/InputStream
#51 = Class          #75          // java/lang/Throwable
#52 = Class          #38          // "[B"
#53 = Class          #83          // java/lang/Exception
#54 = Utf8            Exceptions
#55 = Utf8            stuff
#56 = Utf8            SourceFile
#57 = Utf8            HelloWorldService.java
#58 = Utf8            RuntimeVisibleAnnotations
#59 = Utf8            Lorg/springframework/stereotype/Service;
#60 = NameAndType     #19:#20      // "<init>":()V
#61 = Class           #84          // java/lang/System
#62 = NameAndType     #85:#86      // out:Ljava/io/PrintStream;
#63 = Utf8            Hello, World!
#64 = Class           #87          // java/io/PrintStream
#65 = NameAndType     #88:#89      // println:(Ljava/lang/String;)V
#66 = Utf8            java/lang/Object
#67 = NameAndType     #90:#91      // getClass:()Ljava/lang/Class;
#68 = Class           #92          // java/lang/Class
#69 = NameAndType     #93:#94      //
getResourceAsStream:(Ljava/lang/String;)Ljava/io/InputStream;
#70 = Class           #82          // java/io/InputStream
#71 = NameAndType     #39:#95      // read:([B)I
#72 = Utf8            java/lang/String
#73 = NameAndType     #19:#96      // "<init>":([BII)V
#74 = NameAndType     #97:#20      // close:()V
#75 = Utf8            java/lang/Throwable
#76 = NameAndType     #98:#99      // addSuppressed:(Ljava/lang/Throwable;)V
#77 = Utf8            java/io/IOException
#78 = Utf8            java/lang/RuntimeException
#79 = Class           #83          // java/lang/Exception
#80 = NameAndType     #100:#20     // printStackTrace:()V
#81 = Utf8            com/stackoverflow/documentation/HelloWorldService
#82 = Utf8            java/io/InputStream
#83 = Utf8            java/lang/Exception
#84 = Utf8            java/lang/System
#85 = Utf8            out
#86 = Utf8            Ljava/io/PrintStream;
#87 = Utf8            java/io/PrintStream
#88 = Utf8            println
#89 = Utf8            (Ljava/lang/String;)V
#90 = Utf8            getClass
#91 = Utf8            ()Ljava/lang/Class;
#92 = Utf8            java/lang/Class
#93 = Utf8            getResourceAsStream
#94 = Utf8            (Ljava/lang/String;)Ljava/io/InputStream;
#95 = Utf8            ([B)I
#96 = Utf8            ([BII)V
#97 = Utf8            close
#98 = Utf8            addSuppressed
#99 = Utf8            (Ljava/lang/Throwable;)V
#100 = Utf8           printStackTrace
{
  public com.stackoverflow.documentation.HelloWorldService();
  descriptor: ()V
  flags: ACC_PUBLIC
  Code:
    stack=1, locals=1, args_size=1
      0: aload_0
      1: invokespecial #1          // Method java/lang/Object."<init>":()V
      4: return

```

```

LineNumberTable:
  line 10: 0
LocalVariableTable:
  Start  Length  Slot  Name      Signature
      0       5     0  this      Lcom/stackoverflow/documentation/HelloWorldService;

public void sayHello();
descriptor: ()V
flags: ACC_PUBLIC
Code:
  stack=2, locals=1, args_size=1
    0: getstatic      #2                // Field
java/lang/System.out:Ljava/io/PrintStream;
    3: ldc              #3                // String Hello, World!
    5: invokevirtual #4                // Method
java/io/PrintStream.println:(Ljava/lang/String;)V
    8: return
LineNumberTable:
  line 13: 0
  line 14: 8
LocalVariableTable:
  Start  Length  Slot  Name      Signature
      0       9     0  this      Lcom/stackoverflow/documentation/HelloWorldService;

private java.lang.Object[] pvtMethod(java.util.List<java.lang.String>);
descriptor: (Ljava/util/List;) [Ljava/lang/Object;
flags: ACC_PRIVATE
Code:
  stack=4, locals=2, args_size=2
    0: iconst_1
    1: anewarray        #5                // class java/lang/Object
    4: dup
    5: iconst_0
    6: aload_1
    7: aastore
    8: areturn
LineNumberTable:
  line 17: 0
LocalVariableTable:
  Start  Length  Slot  Name      Signature
      0       9     0  this      Lcom/stackoverflow/documentation/HelloWorldService;
      0       9     1  strings    Ljava/util/List;
LocalVariableTypeTable:
  Start  Length  Slot  Name      Signature
      0       9     1  strings    Ljava/util/List<Ljava/lang/String>;
Signature: #34                //
(Ljava/util/List<Ljava/lang/String>;) [Ljava/lang/Object;

protected java.lang.String tryCatchResources(java.lang.String) throws java.io.IOException;
descriptor: (Ljava/lang/String;)Ljava/lang/String;
flags: ACC_PROTECTED
Code:
  stack=5, locals=10, args_size=2
    0: aload_0
    1: invokevirtual #6                // Method
java/lang/Object.getClass:()Ljava/lang/Class;
    4: aload_1
    5: invokevirtual #7                // Method
java/lang/Class.getResourceAsStream:(Ljava/lang/String;)Ljava/io/InputStream;
    8: astore_2
    9: aconst_null

```

```

10: astore_3
11: sipush      8192
14: newarray    byte
16: astore      4
18: aload_2
19: aload       4
21: invokevirtual #8          // Method java/io/InputStream.read:([B)I
24: istore      5
26: new         #9          // class java/lang/String
29: dup
30: aload       4
32: iconst_0
33: iload       5
35: invokespecial #10         // Method java/lang/String.<init>:([BII)V
38: astore      6
40: aload_2
41: ifnull      70
44: aload_3
45: ifnull      66
48: aload_2
49: invokevirtual #11        // Method java/io/InputStream.close:()V
52: goto        70
55: astore      7
57: aload_3
58: aload       7
60: invokevirtual #13        // Method
java/lang/Throwable.addSuppressed:(Ljava/lang/Throwable;)V
63: goto        70
66: aload_2
67: invokevirtual #11        // Method java/io/InputStream.close:()V
70: aload       6
72: areturn
73: astore      4
75: aload       4
77: astore_3
78: aload       4
80: athrow
81: astore      8
83: aload_2
84: ifnull      113
87: aload_3
88: ifnull      109
91: aload_2
92: invokevirtual #11        // Method java/io/InputStream.close:()V
95: goto        113
98: astore      9
100: aload_3
101: aload       9
103: invokevirtual #13        // Method
java/lang/Throwable.addSuppressed:(Ljava/lang/Throwable;)V
106: goto        113
109: aload_2
110: invokevirtual #11        // Method java/io/InputStream.close:()V
113: aload       8
115: athrow
116: astore_2
117: aload_2
118: invokevirtual #16        // Method
java/lang/Exception.printStackTrace:()V
121: aload_2
122: athrow

```

Exception table:

from	to	target	type
48	52	55	Class java/lang/Throwable
11	40	73	Class java/lang/Throwable
11	40	81	any
91	95	98	Class java/lang/Throwable
73	83	81	any
0	70	116	Class java/io/IOException
0	70	116	Class java/lang/RuntimeException
73	116	116	Class java/io/IOException
73	116	116	Class java/lang/RuntimeException

LineNumberTable:

line 21: 0
line 22: 11
line 23: 18
line 24: 26
line 25: 40
line 21: 73
line 25: 81
line 26: 117
line 27: 121

LocalVariableTable:

Start	Length	Slot	Name	Signature
18	55	4	bytes	[B
26	47	5	read	I
9	107	2	inputStream	Ljava/io/InputStream;
117	6	2	e	Ljava/lang/Exception;
0	123	0	this	Lcom/stackoverflow/documentation/HelloWorldService;
0	123	1	filename	Ljava/lang/String;

StackMapTable: number_of_entries = 9

frame_type = 255 /* full_frame */
offset_delta = 55
locals = [class com/stackoverflow/documentation/HelloWorldService, class
java/lang/String, class java/io/InputStream, class java/lang/Throwable, class "[B", int, class
java/lang/String]
stack = [class java/lang/Throwable]
frame_type = 10 /* same */
frame_type = 3 /* same */
frame_type = 255 /* full_frame */
offset_delta = 2
locals = [class com/stackoverflow/documentation/HelloWorldService, class
java/lang/String, class java/io/InputStream, class java/lang/Throwable]
stack = [class java/lang/Throwable]
frame_type = 71 /* same_locals_1_stack_item */
stack = [class java/lang/Throwable]
frame_type = 255 /* full_frame */
offset_delta = 16
locals = [class com/stackoverflow/documentation/HelloWorldService, class
java/lang/String, class java/io/InputStream, class java/lang/Throwable, top, top, top, top,
class java/lang/Throwable]
stack = [class java/lang/Throwable]
frame_type = 10 /* same */
frame_type = 3 /* same */
frame_type = 255 /* full_frame */
offset_delta = 2
locals = [class com/stackoverflow/documentation/HelloWorldService, class
java/lang/String]
stack = [class java/lang/Exception]
Exceptions:
throws java.io.IOException

```

void stuff();
  descriptor: ()V
  flags:
  Code:
    stack=2, locals=1, args_size=1
    0: getstatic      #2                // Field
java/lang/System.out:Ljava/io/PrintStream;
    3: ldc            #17               // String stuff
    5: invokevirtual #4                // Method
java/io/PrintStream.println:(Ljava/lang/String;)V
    8: return
  LineNumberTable:
    line 32: 0
    line 33: 8
  LocalVariableTable:
    Start  Length  Slot  Name   Signature
        0       9     0   this   Lcom/stackoverflow/documentation/HelloWorldService;
}
SourceFile: "HelloWorldService.java"
RuntimeVisibleAnnotations:
  0: #59()

```

オンラインでアセンブルとコンパイルを <https://riptutorial.com/ja/java/topic/2318/> アセンブルとコンパイル

Examples

カスタム通貨を

クラスパスに追加するJAR

- javax.money:money-api:1.0 JSR354 おと API
- org.javamoney:moneta:1.0 リファレンス
- javax.annotation-api:1.2.0 で される の

```
// Let's create non-ISO currency, such as bitcoin

// At first, this will throw UnknownCurrencyException
MonetaryAmount moneys = Money.of(new BigDecimal("0.1"), "BTC");

// This happens because bitcoin is unknown to default currency
// providers
System.out.println(Monetary.isCurrencyAvailable("BTC")); // false

// We will build new currency using CurrencyUnitBuilder provided by org.javamoney.moneta
CurrencyUnit bitcoin = CurrencyUnitBuilder
    .of("BTC", "BtcCurrencyProvider") // Set currency code and currency provider name
    .setDefaultFractionDigits(2)      // Set default fraction digits
    .build(true);                     // Build new currency unit. Here 'true' means
                                     // currency unit is to be registered and
                                     // accessible within default monetary context

// Now BTC is available
System.out.println(Monetary.isCurrencyAvailable("BTC")); // True
```

オンラインで通貨とマネーを <https://riptutorial.com/ja/java/topic/8359/通貨とマネー>

き

は、のののとをにします。それらはのベクトルにしています。のはにっており、としてします。には、 `int` や `Object` などのプリミティブのデータをすることができます。

- `ArrayType[] myArray;` // をする
- `ArrayType myArray[];` // のなあまりではなく、されません
- `ArrayType[][][] myArray;` // ジグザグをにする[]をりす
- `ArrayType myVar = myArray[index];` // インデックスのへのアクセスみみ
- `myArray[index] = value;` // のindexにををる
- `ArrayType[] myArray = new ArrayType[arrayLength];` // の
- `int[] ints = {1, 2, 3};` // がされた。さは、されたのからされます。 `{[value1 [, value2 *]}`
- `new int[]{4, -5, 6}` // Can be used as argument, without a local variable
 - `int[] ints = new int[3];` // same as `{0, 0, 0}`
 - `int[][] ints = {{1, 2}, {3}, null};` // の。 `int []` は `Object` をしますしたがって `anyType []` もです。 `null` はなです。

パラメーター

パラメータ	
<code>ArrayType</code>	の。これはプリミティブ <code>int</code> 、 <code>long</code> 、 <code>byte</code> または <code>Objects String</code> 、 <code>MyObject</code> などです。
	インデックスとは、のオブジェクトのをします。
さ	されるすべてのには、されたさのセットがです。これは、の <code>new int[3]</code> をするとき、または <code>{1, 2, 3}</code> をするときのうちにわれます。

Examples

のと

なケース

```
int[]    numbers1 = new int[3];           // Array for 3 int values, default value is 0
int[]    numbers2 = { 1, 2, 3 };          // Array literal of 3 int values
int[]    numbers3 = new int[] { 1, 2, 3 }; // Array of 3 int values initialized
int[][]  numbers4 = { { 1, 2 }, { 3, 4, 5 } }; // Jagged array literal
int[][]  numbers5 = new int[5][];         // Jagged array, one dimension 5 long
int[][]  numbers6 = new int[5][4];        // Multidimensional array: 5x4
```

は、のプリミティブまたはをにしてできます。

```
float[]  boats = new float[5];           // Array of five 32-bit floating point numbers.
```

```
double[] header = new double[] { 4.56, 332.267, 7.0, 0.3367, 10.0 };
// Array of five 64-bit floating point numbers.
String[] theory = new String[] { "a", "b", "c" };
// Array of three strings (reference type).
Object[] dArt = new Object[] { new Object(), "We love Stack Overflow.", new Integer(3) };
// Array of three Objects (reference type).
```

のでは、されたのサブタイプがでできることにしてください。

ユーザーのは、プリミティブとにできます

```
UserDefinedClass[] udType = new UserDefinedClass[5];
```

コレクション、およびストリーム

Java SE 1.2

```
// Parameters require objects, not primitives

// Auto-boxing happening for int 127 here
Integer[] initial = { 127, Integer.valueOf( 42 ) };
List<Integer> toList = Arrays.asList( initial ); // Fixed size!

// Note: Works with all collections
Integer[] fromCollection = toList.toArray( new Integer[toList.size()] );

//Java doesn't allow you to create an array of a parameterized type
List<String>[] list = new ArrayList<String>[2]; // Compilation error!
```

Java SE 8

```
// Streams - JDK 8+
Stream<Integer> toStream = Arrays.stream( initial );
Integer[] fromStream = toStream.toArray( Integer[]::new );
```

イントロ

は、のプリミティブまたはオブジェクトインスタンスへのをするデータです。

のはとされ、はそのインデックスによってアクセスされます。がされると、のさがされます。

```
int size = 42;
int[] array = new int[size];
```

のサイズは、ににされます。はできません。にサイズをにするがあるは、わりに [ArrayList](#) などの [Collection](#) クラスををするがあります。ArrayList はにをし、しいをりて、いからをコピーすることによってサイズをサポートします。

がプリミティブの、つまり

```
int[] array1 = { 1,2,3 };
int[] array2 = new int[10];
```

はにされます。の array2 ようにイニシャライザがない、にりてられるデフォルトは0ゼロです。

がオブジェクトの、

```
SomeClassOrInterface[] array = new SomeClassOrInterface[10];
```

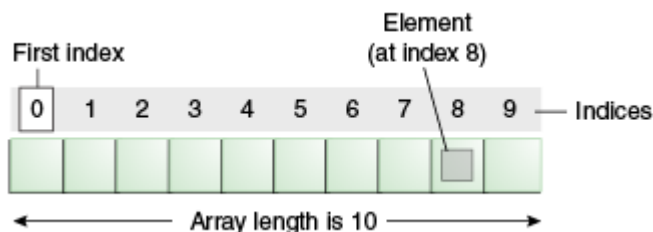
にはSomeClassOrInterfaceのオブジェクトへのがされます。これらのは、のインスタンスをすることが出来るSomeClassOrInterface またはクラスののサブクラスはインターフェイスのクラスSomeClassOrInterface。にがない、 nullのデフォルトがにりてられnull。

すべてのであるのでint -indexed、アレイのサイズは、でなければならないint。のサイズはlongとしてすることはできません。

```
long size = 23L;
int[] array = new int[size]; // Compile-time error:
                             // incompatible types: possible lossy conversion from
                             // long to int
```

は0からまるインデックスシステムをします。つまり、インデックスは0からまり、length - 1します。

たとえば、のサイズが10をしています。ここでは、のインデックス1あり、のインデックス10にあるのではなく、のインデックス0あり、のインデックス9にあります。



のへのアクセスはのにわれます。つまり、ののへのアクセスには、2の、3のなどをアクセスするのとじコストがかかります。

Javaには、リテラルやコンストラクターのなど、のとのがいくつかされています。new Type[length]コンストラクタをにしてをにする、はのデフォルトでされます。

- 0のためのプリミティブ byte、short、int、long、float、およびdouble。
- charの'\u0000' 。
- booleanのはfalse
- のは null

プリミティブのと

```
int[] array1 = new int[] { 1, 2, 3 }; // Create an array with new operator and
// array initializer.
int[] array2 = { 1, 2, 3 }; // Shortcut syntax with array initializer.
int[] array3 = new int[3]; // Equivalent to { 0, 0, 0 }
int[] array4 = null; // The array itself is an object, so it
// can be set as null.
```

をにする、[]はののろ、またはののののろ、またはそのにされます。

```
int array5[]; /* equivalent to */ int[] array5;
int a, b[], c[][]; /* equivalent to */ int a; int[] b; int[][] c;
int[] a, b[]; /* equivalent to */ int[] a; int[][] b;
int a, []b, c[][]; /* Compilation Error, because [] is not part of the type at beginning
of the declaration, rather it is before 'b'. */
// The same rules apply when declaring a method that returns an array:
int foo()[] { ... } /* equivalent to */ int[] foo() { ... }
```

のでは、のがしく、なくコンパイルおよびできます。しかし、JavaコーディングとGoogle Javaスタイルガイドのでは、のろにでまれたフォームがされません。はタイプをし、タイプとともにされます。メソッドリターンシングネチャにもじものをするがあります。

```
float array[]; /* and */ int foo()[] { ... } /* are discouraged */
```

```
float[] array; /* and */ int[] foo() { ... } /* are encouraged */
```

□□されない□は、□□□の□ろに□□□が□いた□の□□に□□している□□□ユーザーに□□するためのものです。

Javaでは、サイズ0の配列を作ることができます。

```
int[] array = new int[0]; // Compiles and runs fine.
int[] array2 = {};       // Equivalent syntax.
```

しかし、それは□の□□なので、そこから□□を□み□むことも、□り□てもできません□

```
array[0] = 1;          // Throws java.lang.ArrayIndexOutOfBoundsException.
int i = array2[0];    // Also throws ArrayIndexOutOfBoundsException.
```

このような`0`の`0`は`0`は`0`りとして`0`です。そのため、`NullPointerException`つながる`0`のある`null`ではなく、`0`び`0`しコードが`0`を`0`うだけで`0`する`0`があり`null`。

□□の□さは、□でない□□でなければなりません□

```
int[] array = new int[-1]; // Throws java.lang.NegativeArraySizeException
```

□□のサイズは、lengthというpublic finalフィールドを□□して□□できます。

```
System.out.println(array.length); // Prints 0 in this case.
```

array.lengthは、与えられた配列の要素の数を返す。ArrayList.size()とは異なり、ArrayListの現在のサイズを返しますが、与えられた配列のサイズは異なります。

_____の_____と_____

□□□□を□□する□も□□な□□は□のとおりです。

```
int[][] a = new int[2][3];
```

a[0]とa[1] 2つの3つの□さのint□□を□□します。これは、□□□の□□□□□の□□□なCスタイルの□□□に□□により□□ています。

□□に□□して□□することができます□

```
int[][] a = { {1, 2}, {3, 4}, {5, 6} };
```

□□の□□□□のみがサポートされているcとは□なり、□□□□は□じ□さである□□はなく、□□する□□もありません。

```
int[][] a = { {1}, {2, 3}, null };
```

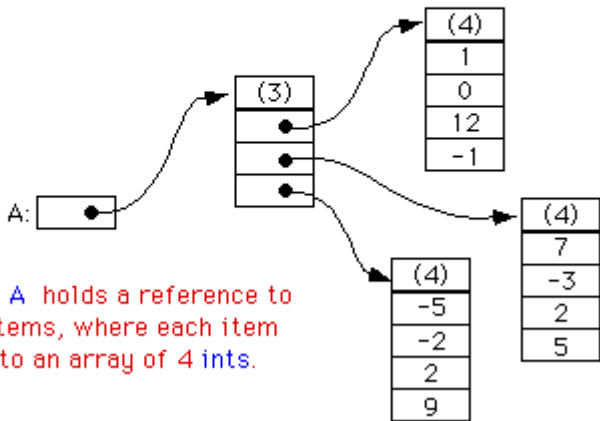
ここでa[0]は10のint型ですがa[1]は20のint型でありa[2]はnullです。このような型は、**ギザギザ型**または**変種型**と呼ばれ、変種型です。Javaの変種型は変種型として扱われます。つまり、array[i][j][k]は((array[i])[j])[k]と等価です。**c++と変種型**、シンタックスarray[i, j]はJavaではサポートされていません。

Javaにおける□□□□□□

A:

1	0	12	-1
7	-3	2	5
-5	-2	2	9

If you create an array `A = new int[3][4]`, you should think of it as a "matrix" with 3 rows and 4 columns.



But in reality, `A` holds a reference to an array of 3 items, where each item is a reference to an array of 4 ints.

ソース - イデオンに開く

String[] の初期化とnull

```
String[] array6 = new String[] { "Laurel", "Hardy" }; // Create an array with new
// operator and array initializer.
String[] array7 = { "Laurel", "Hardy" }; // Shortcut syntax with array
// initializer.
String[] array8 = new String[3]; // { null, null, null }
String[] array9 = null; // null
```

イデオンに開く

`String`のリテラルとプリミティブに包んで、`Object`の配列のためのショートカットは、`Object`でも使えます。

```
Object[] array10 = { new Object(), new Object() };
```

`String`は`Object`であるため、`String`はサブクラスの`Object`として使えますが、`String`を`String`のものにしようすると`ArrayStoreException`がスローされます。

```
Object[] array11 = new String[] { "foo", "bar", "baz" };
array11[1] = "qux"; // fine
array11[1] = new StringBuilder(); // throws ArrayStoreException
```

ショートカットには`Object[]`が使われているため、ショートカットを使うことはできません。

`String`は、`String[] emptyArray = new String[0]`を初期化してゼロで初期化することができます。たとえば、メソッドがオブジェクトの配列を返す場合、`Collection`から`Array`に変換するために、このような`String`がゼロの配列が返されます。

プリミティブと`String`の配列で初期化された例えば`String[] array8 = new String[3]`は、`String`のデフォルトで初期化されます。

ジェネリック配列の初期化とnull

ジェネリッククラスでは、`String`のためにジェネリックの`String`をこのように初期化することはできません。

```
public class MyGenericClass<T> {
    private T[] a;

    public MyGenericClass() {
        a = new T[5]; // Compile time error: generic array creation
```

```

    }
}

```

代わりに、のいずれかをしてできます。これらはチェックのをします。

1. Objectを、ジェネリックにキャストする

```
a = (T[]) new Object[5];
```

これはもですが、になるはとしてObject[]であるため、このメソッドはのをしません。したがって、をこのは、クラスにされていないクラスでのみすることをお勧めします。

2. クラスパラメータでArray.newInstanceをArray.newInstanceと

```
public MyGenericClass(Class<T> clazz) {
    a = (T[]) Array.newInstance(clazz, 5);
}
```

ここでは、Tのクラスをコンストラクタにすがあります。Array.newInstanceのりのはにObjectです。ただし、しくされたはにT[]であるためです。したがって、にできます。

ののりつぶし

Java SE 1.2

Arrays.fill()をして、にじをつをりつぶすことができます。

```
Arrays.fill(array8, "abc"); // { "abc", "abc", "abc" }
```

イデオンにきる

fill()はのされたのにをすることもできます。

```
Arrays.fill(array8, 1, 2, "aaa"); // Placing "aaa" from index 1 to 2.
```

イデオンにきる

Java SE 8

Javaバージョン8のメソッドsetAllとそのConcurrentのparallelSetAllすると、のをされたにできます。これらのメソッドには、インデックスをけり、そのにましいをすジェネレータがされます。

のでは、を、すべてのをそれぞれのインデックスにします。

```
int[] array = new int[5];
Arrays.setAll(array, i -> i); // The array becomes { 0, 1, 2, 3, 4 }.
```

イデオンにきる

のととの

のインデックスのは、0,1,2,3,4,...で、のさインデックスは0からまるよりさくなければなりません。それのは、ArrayIndexOutOfBoundsExceptionがスローされます。

```
int[] array9; // Array declaration - uninitialized
array9 = new int[3]; // Initialize array - { 0, 0, 0 }
array9[0] = 10; // Set index 0 value - { 10, 0, 0 }
array9[1] = 20; // Set index 1 value - { 10, 20, 0 }
array9[2] = 30; // Set index 2 value - { 10, 20, 30 }
```

ArrayListのショートカットを省いてArrayListを初期化することはできません

ArrayListの初期化は、フィールドまたはローカル変数、またはmainメソッドの引数としてしか行うことができないため、ArrayListを初期化してショートカットを省いてArrayListを初期化することはできません。

しかし、新しいArrayListを初期化し、それを古いArrayListを初期化するために使用されるArrayListにリネームすることは可能です。この場合、そのArrayListが初期化するArrayListは初期化されますが、元のArrayListはまったく新しいArrayListになります。これを回避するには、new ArrayListを明示的に使用し、ArrayListに初期化します。

```
// First initialization of array
int[] array = new int[] { 1, 2, 3 };

// Prints "1 2 3 ".
for (int i : array) {
    System.out.print(i + " ");
}

// Re-initializes array to a new int[] array.
array = new int[] { 4, 5, 6 };

// Prints "4 5 6 ".
for (int i : array) {
    System.out.print(i + " ");
}

array = { 1, 2, 3, 4 }; // Compile-time error! Can't re-initialize an array via shortcut
                        // syntax with array initializer.
```

イデオンにリンク

コレクションからのArrayListの初期化

java.util.Collection 2つのメソッドは、コレクションからArrayListを初期化します。

- Object[] toArray()
- <T> T[] toArray(T[] a)

Object[] toArray() は、このように使用できます。

Java SE 5

```
Set<String> set = new HashSet<String>();
set.add("red");
set.add("blue");

// although set is a Set<String>, toArray() returns an Object[] not a String[]
Object[] objectArray = set.toArray();
```

<T> T[] toArray(T[] a) はこのように使用できます。

Java SE 5

```
Set<String> set = new HashSet<String>();
set.add("red");
set.add("blue");

// The array does not need to be created up front with the correct size.
// Only the array type matters. (If the size is wrong, a new array will
```

```
// be created with the same type.)
String[] stringArray = set.toArray(new String[0]);

// If you supply an array of the same size as collection or bigger, it
// will be populated with collection values and returned (new array
// won't be allocated)
String[] stringArray2 = set.toArray(new String[set.size()]);
```

それらの いは、[評価](#)されていない¹と²ただけのことではありません。それらのパフォーマンスも³なる⁴があります⁵しくは、この[パフォーマンスセクション](#)をお⁶みてください。

- Object[] toArray() ベクトル[] arraycopy[] チェックよりもはるかに[]である、arraycopyで[]されるT[] toArray(T[] a)
- T[] toArray(new T[non-zero-size])は[]に[]をゼロアウトする[]がありますが、T[] toArray(new T[0])はそうしません。このような[]は、[]を[]よりも[]く[]び[]す。[]な[]はこちら[] [\[\]の\[\]の\[\]](#)。

Java SE 8

`Stream`の[]が[]されたJava SE 8[]では、`Stream.toArray`メソッドを[]して[]しいArrayを[]するために、コレクションによって[]されたStreamを[]することができます。

```
String[] strings = list.stream().toArray(String[]::new);
```

□としては、□2つの□えから□□した1、 2 □に□□□//「へのArrayListを『□□』 Javaでスタックオーバーフローに□する。

□□□への□□

Java SE 5

Java 1.5.0 では、`Object` の `toString()` をすべてのオブジェクトに継承させることができます。このため、`Arrays.toString(Object[])` または `Arrays.deepToString(Object[])` を呼び出して `Arrays.toString(Object[])` の結果を得ることができます。

```
int[] arr = {1, 2, 3, 4, 5};
System.out.println(Arrays.toString(arr));           // [1, 2, 3, 4, 5]

int[][] arr = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
System.out.println(Arrays.deepToString(arr));      // [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Arrays.toString() メソッドは、Object.toString() メソッドを使用してすべてのオブジェクトのStringを返します。プリミティブ型には、すべてのタイプのオブジェクトに変換できます。例えば

```
public class Cat { /* implicitly extends Object */
    @Override
    public String toString() {
        return "CAT!";
    }
}

Cat[] arr = { new Cat(), new Cat() };
System.out.println(Arrays.toString(arr));           // [CAT!, CAT!]
```

クラスに`toString()`としてオーバーライドされた`toString()`がない場合、`Object`から継承された`toString()`が呼び出されます。これは、`Object`の`toString()`が、`toString()`を返すように定義されています。たとえば、`Object`の`toString()`は、`toString()`を返すように定義されています。


```
public class Dog {
    /* implicitly extends Object */
}

Dog[] arr = { new Dog() };
System.out.println(Arrays.toString(arr));           // [Dog@17ed40e0]
```

からのリストの

`Arrays.asList()` メソッドを使用して、されたの をむ サイズの `List` をすことができます。の `List` は、の とじの になります。

```
String[] stringArray = {"foo", "bar", "baz"};
List<String> stringList = Arrays.asList(stringArray);
```

このリストは、のの ビュー にづいています。つまり、リストをするとがされ、そのもあります。ただし、リストのサイズをすると、のさがされるため、がスローされます。

リストのコピーをするには、`Collection` をとしてる `java.util.ArrayList` のコンストラクタをします。

Java SE 5

```
String[] stringArray = {"foo", "bar", "baz"};
List<String> stringList = new ArrayList<String>(Arrays.asList(stringArray));
```

Java SE 7

Java SE 7 では、<> のをすることができます。これは **ダイヤモンド** とべれます。コンパイラは、コンテキストからをすることができます。つまり、は `ArrayList` のコンストラクタをびすときに `ArrayList` することができ、コンパイルににされます。これは Java **Generics** のである **ダイヤモンド** とべれます。

```
// Using Arrays.asList()

String[] stringArray = {"foo", "bar", "baz"};
List<String> stringList = new ArrayList<>(Arrays.asList(stringArray));

// Using ArrayList.addAll()

String[] stringArray = {"foo", "bar", "baz"};
ArrayList<String> list = new ArrayList<>();
list.addAll(Arrays.asList(stringArray));

// Using Collections.addAll()

String[] stringArray = {"foo", "bar", "baz"};
ArrayList<String> list = new ArrayList<>();
Collections.addAll(list, stringArray);
```

ダイヤモンド についてすべきは、**クラス** ではできないということです。

Java SE 8

```
// Using Streams

int[] ints = {1, 2, 3};
List<Integer> list = Arrays.stream(ints).boxed().collect(Collectors.toList());

String[] stringArray = {"foo", "bar", "baz"};
List<Object> list = Arrays.stream(stringArray).collect(Collectors.toList());
```

Arrays.asList() メソッドの動作に関する注意点

- このメソッドは、Listのインスタンスである、Arrays\$ArrayList クラスのサブクラスArrayList ではなく java.util.ArrayList。として渡られるListは固定サイズです。つまり、Listのサイズまたは内容は UnsupportedOperationException ため、UnsupportedOperationException がスローされます。

```
stringList.add("something"); // throws java.lang.UnsupportedOperationException
```

- 固定サイズにされたListを可変Listコンストラクタに渡すことで、可変Listを作ることができます。これにより、データの可変コピーが作成されます。このコピーは、サイズが可変で、Listの動作をサポートされていません。

```
List<String> modifiableList = new ArrayList<>(Arrays.asList("foo", "bar"));
```

- int[] のようなプリミティブ配列の <T> List<T> asList(T... a) を呼び出すと、List の要素ではなく **int のプリミティブ値だけが** List<int[]> ソースの要素となる

この動作の理由は、ジェネリック型パラメータの代わりにプリミティブ型を使えないため、このメソッドはジェネリック型パラメータがプリミティブ型に置き換えられます。プリミティブ型をListに渡すには、まず、プリミティブ型を包装するラッパークラスの要素にしますつまり、int[] ではなく Integer[] Arrays.asList() を呼び出します。

したがって、これは false

```
int[] arr = {1, 2, 3}; // primitive array of int
System.out.println(Arrays.asList(arr).contains(1));
```

デモを見る

つまり、これは true

```
Integer[] arr = {1, 2, 3}; // object array of Integer (wrapper for int)
System.out.println(Arrays.asList(arr).contains(1));
```

デモを見る

つまり Integer[] として渡られるので、これも true

```
System.out.println(Arrays.asList(1,2,3).contains(1));
```

デモを見る

配列とジグザグ

2次元の配列をジグザグに渡すことは可能です。2次元の配列を渡すことによってアクセスされる代わりに、2次元の配列を渡すことによって、2次元の配列にアクセスします。

2次元の配列は、2次元の配列の宣言に渡すことで渡すことができます。たとえば、2次元のint配列を渡すには、2次元にint[][]などの配列のブラケットセットを渡します。これは、3次元 int[][][] などでも可能です。

3次元の2次元配列を渡すには

```
int rows = 3;
int columns = 3;
int[][] table = new int[rows][columns];
```

この配列を渡して2次元にインデックスを付け、値を渡すことができます。渡されていない値は0のデフォルト値です。この配列はintの値は0です。

```
table[0][0] = 0;
table[0][1] = 1;
table[0][2] = 2;
```

[][]に[][]をインスタンス[]し、[][][][]を[][]することもできます。これらは、より[][]にギザギザアレイと[]ばれます。

```
int[][] nonRect = new int[4][];
```

ジャグ[][]のいずれかのディメンションを[][]することは[][]であるが、それは[][]されている[][]があり、[]のレベルだということに[][]することが[][]です。

```
// valid
String[][] employeeGraph = new String[30][];

// invalid
int[][] unshapenMatrix = new int[][10];

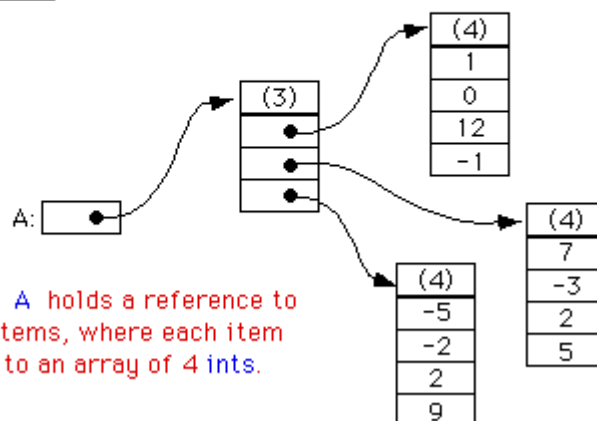
// also invalid
int[][][] misshapenGrid = new int[100][][10];
```

Javaで[][][][]を[][]する[][]

A:

1	0	12	-1
7	-3	2	5
-5	-2	2	9

If you create an array `A = new int[3][4]`, you should think of it as a "matrix" with 3 rows and 4 columns.



ソース [http : //math.hws.edu/eck/cs124/javanotes3/c8/s5.html](http://math.hws.edu/eck/cs124/javanotes3/c8/s5.html)

ギザギザの[][] リテラルの[][]

[][][]とギザギザ[][]は、リテラル[]で[][]することもできます。[][]は、2x3のint[][]を[][]して[]ります

```
int[][] table = {
    {1, 2, 3},
    {4, 5, 6}
};
```

[][] ギザギザのサブ[][]はnullでもかまいません。たとえば、[]のコードでは、[][]のサブ[][]がnull、2[][]のサブ[][]の[]さが0であり、3[][]のサブ[][]の[]さが1で、[][]の[][][]が2つの[]さの[][]である2[][]int[][]が[][]され、

```
int[][] table = {
    null,
    {},
};
```

```

    {1},
    {1,2}
};

```

[][][] の []、そのインデックスによって [][][] の [] を [] することができます。

```

int[][][] arr = new int[3][3][3];
int[][] arr1 = arr[0]; // get first 3x3-dimensional array from arr
int[] arr2 = arr1[0]; // get first 3-dimensional array from arr1
int[] arr3 = arr[0]; // error: cannot convert from int[][] to int[]

```

ArrayIndexOutOfBoundsException

[ArrayIndexOutOfBoundsException](#) は、[] の [] しないインデックスがアクセスされているときにスローされます。

[] は 0 から `array.length - 1` インデックスであるため、[] の [] のインデックスは 0 で、[] の [] のインデックスは [] の [] から 1 いた [] ですつまり `array.length - 1` []。

したがって、インデックス `i` による [][][] の [] は、`0 <= i < array.length` があります。そうでない []、`ArrayIndexOutOfBoundsException` がスローされます。

[] のコードは、`ArrayIndexOutOfBoundsException` がスローされる [] な [] です。

```

String[] people = new String[] { "Carol", "Andy" };

// An array will be created:
// people[0]: "Carol"
// people[1]: "Andy"

// Notice: no item on index 2. Trying to access it triggers the exception:
System.out.println(people[2]); // throws an ArrayIndexOutOfBoundsException.

```

[][]

```

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2
    at your.package.path.method(YourClass.java:15)

```

アクセスされている [] なインデックスも [] に [] まれています [] この [] では 2 []。この [] は [] の [] を [] つけるのに [] ちます。

これを [] けるには、インデックスが [] の [] [] にあることを [] するだけです。

```

int index = 2;
if (index >= 0 && index < people.length) {
    System.out.println(people[index]);
}

```

[] の [] さの []

[] は、[] された [] の [] のサイズまで [] するためのスペースを [] するオブジェクトです。[] のサイズは、[] の [] [] には [] できません。

```

int[] arr1 = new int[0];
int[] arr2 = new int[2];
int[] arr3 = new int[]{1, 2, 3, 4};
int[] arr4 = {1, 2, 3, 4, 5, 6, 7};

```

```
int len1 = arr1.length; // 0
int len2 = arr2.length; // 2
int len3 = arr3.length; // 4
int len4 = arr4.length; // 7
```

のlengthフィールドには、のサイズがされます。これはfinalフィールドであり、することはできません。

このコードは、のlengthとがするオブジェクトののをします。

```
public static void main(String[] args) {
    Integer arr[] = new Integer[] {1,2,3,null,5,null,7,null,null,null,11,null,13};

    int arrayLength = arr.length;
    int nonEmptyElementsCount = 0;

    for (int i=0; i<arrayLength; i++) {
        Integer arrElt = arr[i];
        if (arrElt != null) {
            nonEmptyElementsCount++;
        }
    }

    System.out.println("Array 'arr' has a length of "+arrayLength+"\n"
        + "and it contains "+nonEmptyElementsCount+" non-empty values");
}
```

の

```
Array 'arr' has a length of 13
and it contains 7 non-empty values
```

のの

はequals() およびhashCode() のをjava.lang.Objectからしているため、equals()はまったくじオブジェクトとしたのみにのみtrueをします。それらのについてがしいかどうかをするには、java.util.Arrays.equalsをします。これは、すべてののにしてオーバーロードされます。

```
int[] a = new int[]{1, 2, 3};
int[] b = new int[]{1, 2, 3};
System.out.println(a.equals(b)); //prints "false" because a and b refer to different objects
System.out.println(Arrays.equals(a, b)); //prints "true" because the elements of a and b have the same values
```

がである、Arrays.equals()はのにしてequals()をびしてをします。に、がそれぞれがの、IDのがされます。をするには、のようにArrays.deepEquals()にしてArrays.deepEquals()。

```
int a[] = { 1, 2, 3 };
int b[] = { 1, 2, 3 };

Object[] aObject = { a }; // aObject contains one element
Object[] bObject = { b }; // bObject contains one element

System.out.println(Arrays.equals(aObject, bObject)); // false
System.out.println(Arrays.deepEquals(aObject, bObject)); // true
```

セットとマップではequals()とhashCode()されるため、はにsetやマップキーとしてはにちません。のにしてequals()とhashCode()をするヘルパークラスにラップするか、ListインスタンスににしてListをします。

ストリームへの

Java SE 8

オブジェクトのをStreamにする

```
String[] arr = new String[] { "str1", "str2", "str3" };
Stream<String> stream = Arrays.stream(arr);
```

`Arrays.stream()` を使してプリミティブのをStreamにすると、はStreamのプリミティブなにされます。

```
int[] intArr = {1, 2, 3};
IntStream intStream = Arrays.stream(intArr);
```

Streamをののすることもできます。インデックスはインクルーシブで、インデックスはです。

```
int[] values = {1, 2, 3, 4};
IntStream intStream = Arrays.stream(values, 2, 4);
```

Streamクラスには、`Arrays.stream()` とのメソッド `Stream.of()` があります。いは、`Stream.of()` は `varargs` パラメータを
するため、のようなのがです。

```
Stream<Integer> intStream = Stream.of(1, 2, 3);
Stream<String> stringStream = Stream.of("1", "2", "3");
Stream<Double> doubleStream = Stream.of(new Double[]{1.0, 2.0});
```

の

enhanced forループまたはインデックスを、をすることができます。

```
int[] array = new int[10];

// using indices: read and write
for (int i = 0; i < array.length; i++) {
    array[i] = i;
}
```

Java SE 5

```
// extended for: read only
for (int e : array) {
    System.out.println(e);
}
```

ここでは、のIteratorをすることはないが、ライブラリを利用してIterableオブジェクトをするためのリストに
にできることにIterableすることはIterableです。

ボックスされたの、`Arrays.asList` を使します

```
Integer[] boxed = {1, 2, 3};
Iterable<Integer> boxedIt = Arrays.asList(boxed); // list-backed iterable
Iterator<Integer> fromBoxed1 = boxedIt.iterator();
```

プリミティブ java 8では、ストリームにこのでは `Arrays.stream` - > `IntStream` を使します。

```
int[] primitives = {1, 2, 3};
IntStream primitiveStream = Arrays.stream(primitives); // list-backed iterable
```

```
PrimitiveIterator.OfInt fromPrimitive1 = primitiveStream.iterator();
```

ストリームJava 8なしをできないは、Googleの[グア](#)ライブラリをすることが出来ます。

```
Iterable<Integer> fromPrimitive2 = Ints.asList(primitives);
```

2Dまたはそれでは、のをややなですることが出来ます。

```
int[][] array = new int[10][10];

for (int indexOuter = 0; indexOuter < array.length; indexOuter++) {
    for (int indexInner = 0; indexInner < array[indexOuter].length; indexInner++) {
        array[indexOuter][indexInner] = indexOuter + indexInner;
    }
}
```

Java SE 5

```
for (int[] numbers : array) {
    for (int value : numbers) {
        System.out.println(value);
    }
}
```

インデックススペースのループをせずにののなにすることはです。

もちろん、インデックスをってするときwhileやdo-whileループをうことも出来ます。

すべきは、インデックスをするは、インデックスが0からarray.length - 1 をむのにあることをしてください。のさについてハードコードされたをしないでください。そうでなければ、のさはわかりますが、ハードコードされたはわかりません。

```
int[] numbers = {1, 2, 3, 4};

public void incrementNumbers() {
    // DO THIS :
    for (int i = 0; i < numbers.length; i++) {
        numbers[i] += 1; //or this: numbers[i] = numbers[i] + 1; or numbers[i]++;
    }

    // DON'T DO THIS :
    for (int i = 0; i < 4; i++) {
        numbers[i] += 1;
    }
}
```

なをしてインデックスをするのではなく、インデックスをしてをい、なるがなはそれらをするのがです。

```
public void fillArrayWithDoubleIndex(int[] array) {
    // DO THIS :
```

```

for (int i = 0; i < array.length; i++) {
    array[i] = i * 2;
}

// DON'T DO THIS :
int doubleLength = array.length * 2;
for (int i = 0; i < doubleLength; i += 2) {
    array[i / 2] = i;
}
}

```

配列の要素へのアクセス

```

int[] array = {0, 1, 1, 2, 3, 5, 8, 13};
for (int i = array.length - 1; i >= 0; i--) {
    System.out.println(array[i]);
}

```

冗長なコードを短くしてコードの可読性を高める

コードを短くする代わりに配列を操作すると、コードがきれいになります。配列の要素にアクセスして値が取得されるようにすることができます。

```

// we want to print out all of these
String name = "Margaret";
int eyeCount = 16;
double height = 50.2;
int legs = 9;
int arms = 5;

// copy-paste approach:
System.out.println(name);
System.out.println(eyeCount);
System.out.println(height);
System.out.println(legs);
System.out.println(arms);

// temporary array approach:
for(Object attribute : new Object[]{name, eyeCount, height, legs, arms})
    System.out.println(attribute);

// using only numbers
for(double number : new double[]{eyeCount, legs, arms, height})
    System.out.println(Math.sqrt(number));

```

このコードは、ループが実行されるたびに値が取得され、プリミティブ型がメモリにコピーされてしまうため、パフォーマンスのセクションでは避けないでください。

配列のコピー

Javaは、配列をコピーするいくつかの方法を提供します。

forループ

```

int[] a = { 4, 1, 3, 2 };
int[] b = new int[a.length];
for (int i = 0; i < a.length; i++) {

```



```
b[i] = a[i];  
}
```

このオプションをプリミティブの代わりにオブジェクトにすると、コピーの代わりにのをコピーがめられます。

Object.clone

はJavaのObjectため、`Object.clone()`できます。

```
int[] a = { 4, 1, 3, 2 };  
int[] b = a.clone(); // [4, 1, 3, 2]
```

のObject.cloneメソッドは、のコピーをします。つまり、ソースとをコピーするのへのをします。

Arrays.copyOf

`java.util.Arrays`はのコピーをのにのするをします。ここではのいいです。

```
int[] a = {4, 1, 3, 2};  
int[] b = Arrays.copyOf(a, a.length); // [4, 1, 3, 2]
```

`Arrays.copyOf`は、ののことができるオーバーロードもされています。

```
Double[] doubles = { 1.0, 2.0, 3.0 };  
Number[] numbers = Arrays.copyOf(doubles, doubles.length, Number[].class);
```

System.arraycopy

```
public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int  
length)
```

されたからまる、されたソースのを、コピーのされたにコピーします。

の

```
int[] a = { 4, 1, 3, 2 };  
int[] b = new int[a.length];  
System.arraycopy(a, 0, b, 0, a.length); // [4, 1, 3, 2]
```

Arrays.copyOfRange

にのをコピーするためにされますが、をのようにコピーすることもできます。

```
int[] a = { 4, 1, 3, 2 };  
int[] b = Arrays.copyOfRange(a, 0, a.length); // [4, 1, 3, 2]
```

キャスト

はオブジェクトですが、そのはまれるオブジェクトのによってされます。したがって、`A[]`を`T[]`にキャスト`A[]`ことはできませんが、の`A[]`メンバを`T`オブジェクトにキャストするがあります。の

```
public static <T, A> T[] castArray(T[] target, A[] array) {  
    for (int i = 0; i < array.length; i++) {  
        target[i] = (T) array[i];  
    }  
}
```

```

    }
    return target;
}

```

したがって、`A[]`が与えられたと`A[]`。

```

T[] target = new T[array.Length];
target = castArray(target, array);

```

Java SEは、この目的のために`Arrays.copyOf(original, newLength, newType)`メソッドを提供します。

```

Double[] doubles = { 1.0, 2.0, 3.0 };
Number[] numbers = Arrays.copyOf(doubles, doubles.length, Number[].class);

```

からを

Javaはからをするための`java.util.Arrays`なメソッドを提供しません。これをするには、のをなしでいかにコピーするか、またはをのに置いて行うことができます。

ArrayListの

を`java.util.List`にし、をとして、リストをのようににすることができます。

```

String[] array = new String[]{"foo", "bar", "baz"};

List<String> list = new ArrayList<>(Arrays.asList(array));
list.remove("foo");

// Creates a new array with the same size as the list and copies the list
// elements to it.
array = list.toArray(new String[list.size()]);

System.out.println(Arrays.toString(array)); //[bar, baz]

```

System.arraycopyの

`System.arraycopy()`をとして、ののコピーをし、なをすることができます。にをします。

```

int[] array = new int[] { 1, 2, 3, 4 }; // Original array.
int[] result = new int[array.length - 1]; // Array which will contain the result.
int index = 1; // Remove the value "2".

// Copy the elements at the left of the index.
System.arraycopy(array, 0, result, 0, index);
// Copy the elements at the right of the index.
System.arraycopy(array, index + 1, result, index, array.length - index - 1);

System.out.println(Arrays.toString(result)); //[1, 3, 4]

```

Apache Commons Langの

をにするには、`Apache Commons Lang`ライブラリ、に`ArrayUtils`クラスのメソッド`removeElement()`を使用できます。にをします。

```

int[] array = new int[]{1,2,3,4};

```

```
array = ArrayUtils.removeElement(array, 2); //remove first occurrence of 2
System.out.println(Arrays.toString(array)); //[1, 3, 4]
```

の

オブジェクトはです。つまり、IntegerはNumberサブクラスと同じように、Integer[]はNumber[]サブクラスです。これはのようにえるかもしれませんが、くべきるいにつながります

```
Integer[] integerArray = {1, 2, 3};
Number[] numberArray = integerArray; // valid
Number firstElement = numberArray[0]; // valid
numberArray[0] = 4L; // throws ArrayStoreException at runtime
```

Integer[]はNumber[]サブクラスですが、Integerのみをでき、Longをりてようとするとがスローされます。このはのであり、わりにListをすることでできます。

```
List<Integer> integerList = Arrays.asList(1, 2, 3);
//List<Number> numberList = integerList; // compile error
List<? extends Number> numberList = integerList;
Number firstElement = numberList.get(0);
//numberList.set(0, 4L); // compile error
```

ののサブクラスであるり、すべてのがじをするはありません。

```
interface I {}

class A implements I {}
class B implements I {}
class C implements I {}

I[] array10 = new I[] { new A(), new B(), new C() }; // Create an array with new
// operator and array initializer.

I[] array11 = { new A(), new B(), new C() }; // Shortcut syntax with array
// initializer.

I[] array12 = new I[3]; // { null, null, null }

I[] array13 = new A[] { new A(), new A() }; // Works because A implements I.

Object[] array14 = new Object[] { "Hello, World!", 3.14159, 42 }; // Create an array with
// new operator and array initializer.

Object[] array15 = { new A(), 64, "My String" }; // Shortcut syntax
// with array initializer.
```

どのようにのサイズをしますか

なえはあなたがこれを行うことができないということです。アレイがされると、そのサイズはできません。わりに、なサイズのしいをし、のからしいにをコピーすることによってのみ、のサイズをすることができます。

```
String[] listOfCities = new String[3]; // array created with size 3.
listOfCities[0] = "New York";
listOfCities[1] = "London";
listOfCities[2] = "Berlin";
```

たとえば、のようにされたlistOfCitiesにしいをあるとします。これをうには、がです。

1. サイズ4の新しい配列を宣言し、
2. 古い配列の最初の3つの要素をオフセット0,1,2で新しい配列にコピーし、
3. オフセット3の新しい配列に新しい要素を追加します。

配列を扱うにはさまざまな方法があります。Java 6では、これも可能な方法は次のとおりでした。

```
String[] newArray = new String[listOfCities.length + 1];
System.arraycopy(listOfCities, 0, newArray, 0, listOfCities.length);
newArray[listOfCities.length] = "Sydney";
```

現在のJava 6からは、`Arrays.copyOf`と`Arrays.copyOfRange`がより簡単にこれを行うことができます。

```
String[] newArray = Arrays.copyOf(listOfCities, listOfCities.length + 1);
newArray[listOfCities.length] = "Sydney";
```

アレイをコピーする際の注意点については、別のページをご覧ください。サイズを変更するときは、オリジナルとは異なるサイズの新しいコピーが必要であることに注意してください。

・ 配列のコピー

配列のサイズ変更のより良い方法

配列のように配列のサイズを変更すると、2つの問題があります。

- ・ それは遅いです。配列を大きくするまたは小さくするには、配列のすべての要素をコピーし、新しいオブジェクトを割り当てます。配列が大きければ大きいほど、それは遅くなります。
- ・ 古い配列への参照を保持する"ライブ"配列を保持できる方法があります。

1つの問題は、配列に異なるサイズの配列を保持することです。これは、配列を保持する際にサイズを変更できない配列にのみ適用されます。

もう1つの問題は、Java SEクラスライブラリまたはサードパーティのライブラリによって提供されるデータ構造クラスを保持することです。たとえば、Java SEの「コレクション」フレームワークでは、さまざまなプロパティを持つList API、Set APIおよびMap APIのすべてが提供されています。ArrayListクラスは、配列の柔軟性を失わずに柔軟なサイズ変更を可能しながら、プレーンなO(1)ランダムアクセス、O(1)getおよびset、O(N)ランダムアクセスおよび削除などのパフォーマンスにも優れたArrayList。

ArrayListのサイズ変更の問題は、リサイズでバッキングのサイズを2倍にするという点から来ていますが、一般的なユースケースでは、リサイズするだけです。リストのライフタイムにわたってリサイズすると、インサートごとにO(1)です。プレーンな配列のサイズを変更するときにも同じ問題を解決することができます。

配列の問題を解決する

配列の問題を解決する方法はたくさんあります。このページのスニペットは、配列がどれかであるとみなしています。

```
String[] strings = new String[] { "A", "B", "C" };
int[] ints = new int[] { 1, 2, 3, 4 };
```

さらに、それぞれindexまたはindex2を異なる配列のインデックスに渡しindex。配列が空でない場合は-1をindex。

Arrays.binarySearch 配列がソートされた配列のみ

```
int index = Arrays.binarySearch(strings, "A");
int index2 = Arrays.binarySearch(ints, 1);
```

Arrays.asList 配列プリミティブ配列のみ

```
int index = Arrays.asList(strings).indexOf("A");
int index2 = Arrays.asList(ints).indexOf(1); // compilation error
```

Stream

□□

Java SE 8

```
int index = IntStream.range(0, strings.length)
    .filter(i -> "A".equals(strings[i]))
    .findFirst()
    .orElse(-1); // If not present, gives us -1.
// Similar for an array of primitives
```

ループを□った□□□□

```
int index = -1;
for (int i = 0; i < array.length; i++) {
    if ("A".equals(array[i])) {
        index = i;
        break;
    }
}
// Similar for an array of primitives
```

[org.apache.commons](#)などの□□□のライブラリを□□した□□□□

```
int index = org.apache.commons.lang3.ArrayUtils.contains(strings, "A");
int index2 = org.apache.commons.lang3.ArrayUtils.contains(ints, 1);
```

□□□□□□□□を□□する□が、リスト□で□り□すより□□です。

□□に□□か□まれているかどうかを□べる

□□の□は、□□されたインデックスがゼロ□□であるかどうかを□にテストすることによって、□□に□□が□まれているかどうかをテストするために□□させることができます。

あるいは、いくつかの□□なバリエーションもあります。

```
boolean isPresent = Arrays.asList(strings).contains("A");
```

Java SE 8

```
boolean isPresent = Stream<String>.of(strings).anyMatch(x -> "A".equals(x));
```

```
boolean isPresent = false;
for (String s : strings) {
    if ("A".equals(s)) {
        isPresent = true;
        break;
    }
}

boolean isPresent = org.apache.commons.lang3.ArrayUtils.contains(ints, 4);
```

□□の□べ□え

□□の□べ□えは、[□□ API](#)を□って□□に□うことができます。

```
import java.util.Arrays;
```

```
// creating an array with integers
int[] array = {7, 4, 2, 1, 19};
// this is the sorting part just one function ready to be used
Arrays.sort(array);
// prints [1, 2, 4, 7, 19]
System.out.println(Arrays.toString(array));
```

Stringのソート

Stringはデータではなく、オブジェクトと見られるオブジェクトの配列を扱います。アルファベット順とも見られます。sort()メソッドを呼び出してStringの配列をソートすると、配列はComparableインターフェイスで実装されたオブジェクトにソートされます。

ソートする

```
String[] names = {"John", "Steve", "Shane", "Adam", "Ben"};
System.out.println("String array before sorting : " + Arrays.toString(names));
Arrays.sort(names);
System.out.println("String array after sorting in ascending order : " +
Arrays.toString(names));
```

出力

```
String array before sorting : [John, Steve, Shane, Adam, Ben]
String array after sorting in ascending order : [Adam, Ben, John, Shane, Steve]
```

逆順のソート

```
Arrays.sort(names, 0, names.length, Collections.reverseOrder());
System.out.println("String array after sorting in descending order : " +
Arrays.toString(names));
```

出力

```
String array after sorting in descending order : [Steve, Shane, John, Ben, Adam]
```

オブジェクト配列のソート

オブジェクト配列をソートするには、ソートのオブジェクトを比較するために、すべてのオブジェクトがComparableまたはComparatorどちらかのインタフェースを実装する必要があります。

いずれかのsort(Object[])メソッドを呼び出して配列でオブジェクト配列をソートできますが、配列のすべてのオブジェクトがComparableでComparableを実装する必要があります。

さらに、e1.compareTo(e2)はオブジェクトe1とe2に対してClassCastExceptionをスローしてはいけません。あるいは、配列の要素のようにsort(T[], Comparator)メソッドを呼び出してカスタムオブジェクト配列をソートすることもできます。

```
// How to Sort Object Array in Java using Comparator and Comparable
Course[] courses = new Course[4];
courses[0] = new Course(101, "Java", 200);
courses[1] = new Course(201, "Ruby", 300);
courses[2] = new Course(301, "Python", 400);
courses[3] = new Course(401, "Scala", 500);

System.out.println("Object array before sorting : " + Arrays.toString(courses));

Arrays.sort(courses);
System.out.println("Object array after sorting in natural order : " +
Arrays.toString(courses));
```

```
Arrays.sort(courses, new Course.PriceComparator());
System.out.println("Object array after sorting by price : " + Arrays.toString(courses));

Arrays.sort(courses, new Course.NameComparator());
System.out.println("Object array after sorting by name : " + Arrays.toString(courses));
```

□□□

```
Object array before sorting : [#101 Java@200 , #201 Ruby@300 , #301 Python@400 , #401
Scala@500 ]
Object array after sorting in natural order : [#101 Java@200 , #201 Ruby@300 , #301 Python@400
, #401 Scala@500 ]
Object array after sorting by price : [#101 Java@200 , #201 Ruby@300 , #301 Python@400 , #401
Scala@500 ]
Object array after sorting by name : [#101 Java@200 , #301 Python@400 , #201 Ruby@300 , #401
Scala@500 ]
```

プリミティブとボックス□の□の□□の□□

□□によっては、[プリミティブ](#)□から[ボックス](#)□への□□が□□になることがあります。

□□を□□するには、ストリームを□□することができます□Java 8□□□。

Java SE 8

```
int[] primitiveArray = {1, 2, 3, 4};
Integer[] boxedArray =
    Arrays.stream(primitiveArray).boxed().toArray(Integer[]::new);
```

□□バージョンでは、プリミティブ□□を□□して□□□にボックス□された□□にコピーすることができます□

Java SE 8

```
int[] primitiveArray = {1, 2, 3, 4};
Integer[] boxedArray = new Integer[primitiveArray.length];
for (int i = 0; i < primitiveArray.length; ++i) {
    boxedArray[i] = primitiveArray[i]; // Each element is autoboxed here
}
```

□□に、ボックス□された□□は、そのプリミティブの□□する□□に□□できます。

Java SE 8

```
Integer[] boxedArray = {1, 2, 3, 4};
int[] primitiveArray =
    Arrays.stream(boxedArray).mapToInt(Integer::intValue).toArray();
```

Java SE 8

```
Integer[] boxedArray = {1, 2, 3, 4};
int[] primitiveArray = new int[boxedArray.length];
for (int i = 0; i < boxedArray.length; ++i) {
    primitiveArray[i] = boxedArray[i]; // Each element is outboxed here
}
```

オンラインで□□を□む□ <https://riptutorial.com/ja/java/topic/99/□□>

185: アクセス

き

アクセスは、やメソッドのアクセシビリティを するものではありませんが、 なプロパティを します。

Examples

の

Javaのfinalは、メソッド、クラスを できます。3つの なるルールがあります

- を することはできません
- な をオーバーライドすることはできません
- クラスは できません

いプログラミング

いくつかの は などのときに final をマークすることをお勧めします。 えてはいけない がある は、それを にマークする があります。

メソッドのパラメータの は、final キーワードの な 。メソッドが パラメータを しないことを したい は、プロパティを final としてマークします。

```
public int sumup(final List<Integer> ints);
```

これは、sumupメソッドがintsを しないことを して ints。

インナークラスアクセス

あなたの の クラスが にアクセスしたい は、 に final とマークする があります

```
public IPrintName printName(){
    String name;
    return new IPrintName(){
        @Override
        public void printName(){
            System.out.println(name);
        }
    };
}
```

このクラスはコンパイルされず、nameはfinalではありません。

Java SE 8

な は です。これらはローカル であり、 だけ き まれるため、 に することができます。な は クラスからもアクセスできます。

final static

のコードは、ときに に であってもfinal fooないstaticの、staticそれがコンパイルされません。

```
class TestFinal {
    private final static List foo;

    public Test() {
```



```

        foo = new ArrayList();
    }
}

```

これは、もうやりましょう。静的なものは書きません。fooは静的なので、TestFinalクラスのすべてのインスタンスで共有されます。TestFinalクラスの新しいインスタンスが作成されると、そのコンストラクターが呼び出されるため、コンパイラが静的でないfooが書き換えられます。この場合、静的fooを共有する新しいものは、そのいずれかです。

```

class TestFinal {
    private static final List foo = new ArrayList();
    //..
}

```

または別の方法として

```

class TestFinal {
    private static final List foo;
    static {
        foo = new ArrayList();
    }
    //..
}

```

静的なメソッドは、このクラスが、このクラスがそれを共有することを許可していない静的なものを共有することにできます。静的テーブルの概念が静的でないため、静的メソッドよりもです。

Javaのすべてのラッパークラスは、Integer、Longなどのように静的です。これらのクラスのクリエイターは、Integerをこのクラスに静的し、Integerクラスの静的なものを共有することはできません。クラスを静的にするための方法の1つは、サブクラスがメソッドをオーバーライドできないことです。これを防ぐ最も簡単な方法は、クラスをfinalとして宣言することです。

別の

volatileは、マルチスレッドプログラミングで使われます。フィールドをvolatileとすると、フィールドをローカルにキャッシュされていない別のスレッドに知らせるシグナルとなります。さらに、volatileを読み取りと書き込みはアトミックでvolatile longまたはdoubleアクセスはアトミックではないであることが保証されているため、別のスレッドで読み取り/書き込みエラーが起きません。

```

public class MyRunnable implements Runnable
{
    private volatile boolean active;

    public void run(){ // run is called in one thread
        active = true;
        while (active){
            // some code here
        }
    }

    public void stop(){ // stop() is called from another thread
        active = false;
    }
}

```

また

staticキーワードは、クラス、メソッド、またはフィールドで使われ、クラスのインスタンスとは別に存在します。

- 静的フィールドは、クラスのすべてのインスタンスに共有です。これはそれらにアクセスするためのインスタンスを共有とします

せん。

- 静的メソッドは、そのクラスのインスタンスなしで呼び出すことができますが、そのクラスの静的フィールドにのみアクセスできます。
- 静的クラスは別のクラスの静的メソッドを呼び出すことができます。インスタンス化するクラスのインスタンスは存在しません。

```
public class TestStatic
{
    static int staticVariable;

    static {
        // This block of code is run when the class first loads
        staticVariable = 11;
    }

    int nonStaticVariable = 5;

    static void doSomething() {
        // We can access static variables from static methods
        staticVariable = 10;
    }

    void add() {
        // We can access both static and non-static variables from non-static methods
        nonStaticVariable += staticVariable;
    }

    static class StaticInnerClass {
        int number;
        public StaticInnerClass(int _number) {
            number = _number;
        }

        void doSomething() {
            // We can access number and staticVariable, but not nonStaticVariable
            number += staticVariable;
        }

        int getNumber() {
            return number;
        }
    }
}

// Static fields and methods
TestStatic object1 = new TestStatic();

System.out.println(object1.staticVariable); // 11
System.out.println(TestStatic.staticVariable); // 11

TestStatic.doSomething();

TestStatic object2 = new TestStatic();

System.out.println(object1.staticVariable); // 10
System.out.println(object2.staticVariable); // 10
System.out.println(TestStatic.staticVariable); // 10

object1.add();

System.out.println(object1.nonStaticVariable); // 15
System.out.println(object2.nonStaticVariable); // 10
```

```
// Static inner classes
StaticInnerClass object3 = new TestStatic.StaticInnerClass(100);
StaticInnerClass object4 = new TestStatic.StaticInnerClass(200);

System.out.println(object3.getNumber()); // 100
System.out.println(object4.getNumber()); // 200

object3.doSomething();

System.out.println(object3.getNumber()); // 110
System.out.println(object4.getNumber()); // 200
```

□□

□□□は、□□の□□を□し、□□だけをユーザーに□すプロセスです。□□クラスはインスタンス□できません。□□クラスとして□□されたクラスは、クラスの□□のみを□□としています。

```
abstract class Car
{
    abstract void tagLine();
}

class Honda extends Car
{
    void tagLine()
    {
        System.out.println("Start Something Special");
    }
}

class Toyota extends Car
{
    void tagLine()
    {
        System.out.println("Drive Your Dreams");
    }
}
```

□□された

□□□□□□は、□□のメソッドまたは□□のスレッドによるブロックのアクセスを□□するために□□されます。synchronizedとして□□されているメソッドまたはブロックには、1つのスレッドだけが□ることができます。synchronizedキーワードはオブジェクトの□□のロックに□□し、□□メソッドの□□には□□のオブジェクトロックを□ち、□□メソッドはクラスオブジェクトを□□します。□□ブロックを□□しようとするスレッドは、□□にオブジェクトロックを□□する□□があります。

```
class Shared
{
    int i;

    synchronized void SharedMethod()
    {
        Thread t = Thread.currentThread();

        for(int i = 0; i <= 1000; i++)
        {
            System.out.println(t.getName()+" : "+i);
        }
    }
}
```

```

    }

    void SharedMethod2()
    {
        synchronized (this)
        {
            System.out.println("Thais access to current object is synchronize "+this);
        }
    }
}

public class ThreadsInJava
{
    public static void main(String[] args)
    {
        final Shared s1 = new Shared();

        Thread t1 = new Thread("Thread - 1")
        {
            @Override
            public void run()
            {
                s1.SharedMethod();
            }
        };

        Thread t2 = new Thread("Thread - 2")
        {
            @Override
            public void run()
            {
                s1.SharedMethod();
            }
        };

        t1.start();

        t2.start();
    }
}

```

□□□な

□□□□として□□された□□は、オブジェクトの□□□□にシリアル□されません。

```

public transient int limit = 55;    // will not persist
public int b; // will persist

```

strictfp

Java SE 1.2

strictfp□□□は、□□□□□□□に□□されます。この□□□は、□□□□□□□を□□のプラットフォームでより□□□□のあるものにし、□□□□□□□がIEEE 754□□□に□□って□□われ、32ビットおよび64ビットアーキテクチャの□□で□□□□□□□□□□、オーバーフローおよびアンダーフローのエラーを□□します。これは、□□メソッド、□□、またはコンストラクタには□□できません。

```

// strictfp keyword can be applied on methods, classes and interfaces.

strictfp class A{

```

```
strictfp interface M{}  
  
class A{  
    strictfp void m(){}  
}
```

オンラインで「アクセス」を「む」 <https://riptutorial.com/ja/java/topic/4401/>「アクセス」

S. No		Contributors
1	Javaのライブラリをいめる	aa_oo , Aaqib Akhtar , abhinav , Abhishek Jain , Abob , acdcjunior , Adeel Ansari , adsalpha , AER , akhilsk , Akshit Soota , Alex A , alphaloop , altomnr , Amani Kilumanga , AndroidMechanic , Ani Menon , ankit dassor , Ankur Anand , antonio , Arkadiy , Ashish Ahuja , Ben Page , Blachshma , bpoiss , Burkhard , Carlton , Charlie H , Coffeehouse Coder , c1dsEED , Community , Configure , CraftedCart , dabansal , Daksh Gupta , Dan Hulme , Dan Morenus , DarkVl , David G. , David Grinberg , David Newcomb , DeepCoder , Do Nhu Vy , Draken , Durgpal Singh , Dushko Jovanovski , E_net4 , Edvin Tenovimas , Emil Sierżęga , Emre Bolat , enrico.bacis , Eran , explv , fgb , Francesco Menzani , Functino , gargl0may , Gautam Jose , GingerHead , Grzegorz Górkiwicz , iliketocode , иәғәт әізуәх , intboolstring , ipsi , J F , James Taylor , Jason , JavaHopper , Javant , javydreamercsw , Jean Vitor , Jean-François Savard , Jeffrey Brett Coleman , Jeffrey Lin , Jens Schauder , John Fergus , John Riddick , John Slegers , Jojodmo , JonasCz , Jonathan , Jonny Henly , Jorn Vernee , kaartic , Lambda Ninja , LostAvatar , madx , Magisch , Makoto , manetsus , Marc , Mark Adelsberger , Maroun Maroun , Matt , Matt , mayojava , Mitch Talmadge , mnoronha , Mrunal Pagnis , Mukund B , Mureinik , NageN , Nathan Arthur , nevster , Nithanim , Nuri Tasdemir , nyarasha , ochi , OldMcDonald , Onur , Ortomala Lokni , OverCoder , P.J.Meisch , Pavneet_Singh , Petter Friberg , philnate , Phrancis , Pops , ppeterka , Přemysl Šťastný , Pritam Banerjee , Radek Postołowicz , Radouane ROUFID , Rafael Mello , Rakitić , Ram , RamenChef , rekire , René Link , Reut Sharabani , Richard Hamilton , Ronnie Wang , ronnyfm , Ross Drew , RotemDev , Ryan Hilbert , SachinSarawgi , Sanandrea , Sandeep Chatterjee , Sayakiss , ShivBuyya , Shoe , Siguza , solidcell , stackptr , Stephen C , Stephen Leppik , sudo , Sumurai8 , Shadowfa , tbodt , The Coder , ThePhantomGamer , Thisaru Guruge , Thomas Gerot , ThomasThiebaud , ThunderStruct , tonirush , Tushar Mudgal , Unihedron , user1133275 , user124993 , uzaif , Vaibhav Jain , Vakerrian , vasili111 , Victor Stafusa , Vin , VinayVeluri , Vogel612 , vorburger , Wilson , worker_bee , Yash Jain , Yury Fedorov , Zachary David Saunders , Ze Rubeus
2	Apache Commons Lang	Jonathan Barbero
3	BigDecimal	alain.janinm , Christian , Dth , Enigo , ggolding , Harish Gyanani , John Nash , Loris Securo , Łukasz Piaszczyk , Manish Kothari , mszymborski , RamenChef , sudo , xwoker
4	BigInteger	Alek Mieczkowski , Alex Shesterov , Amani Kilumanga , Andrii Abramov , azurefrog , Byte1518 , dimo414 , dorukayhan , Emil Sierżęga , fabian , GPI , Ha. , hd84335 , janos , Kaushal28 , Maarten Bodewes , Makoto , matt freake , Md. Nasir Uddin Bhuiyan , Nufail , Pritam Banerjee , Ruslan Bes , ShivBuyya , Stendika , Vogel612
5	BufferedWriter	Andrii Abramov , fabian , Jorn Vernee , Robin , VatsalSura
6	ByteBuffer	Community , Jon Ericson , Jorn Vernee , Tarık Yılmaz , Tomasz Bawor , victorantunes , Vogel612
7	C ++の	John DiFinì
8	EnumSetクラス	KIRAN KUMAR MATAM

9	Enumマップ	KIRAN KUMAR MATAM
10	Executor、ExecutorService、Threadsプール	Andrii Abramov, Cache Staheli, Fildor, hd84335, Jens Schauder, JonasCz, noscreenname, Olivier Grégoire, philnate, Ravindra babu, Shettyh, Stephen C, Suminda Sirinath S. Dharmasena, sv3k, tones, user1121883, Vlad-HC, Vogel612
11	FileUpload to AWS	Amit Gujarathi
12	FTPファイルプロトコル	Kelvin Kellner
13	HttpURLConnection	Community, Datagrammar, EJP, Inzimam Tariq IT, JonasCz, kiedysktos, Mureinik, NageN, Stephen C, still_learning
14	InputStreamsとOutputStreams	akgren_soar, EJP, Gubbel, J Atkin, Jens Schauder, John Nash, Kip, KIRAN KUMAR MATAM, Matt Clark, Michael, RamenChef, Stephen C, Vogel612
15	Java Edition、バージョン、リリースおよびディストリビューション	Gal Dreiman, screab, Stephen C
16	Java SE 7の	compuhosny, RamenChef
17	Java SE 8の	compuhosny, RamenChef, sun-solar-arrow
18	Java Standard Editionのインストール	4444, Adeel Ansari, ajablonski, akhilsk, Alex A, altomnr, Ani Menon, Anthony Raymond, anuvabl911, Configure, CraftedCart, Emil Sierżęga, Gautam Jose, hd84335, ipsi, Jeffrey Brett Coleman, Lambda Ninja, Nithanim, Radouane ROUFID, Rakitić, ronnyfm, Sanandrea, Sandeep Chatterjee, sohnryang, Stephen C, Shadowfa, tonirush, Walery Strauch, Ze Rubeus
19	java.util.Objectsクラス	mnoronha, RamenChef, Stephen C
20	JavaBean	foxt7ot, J. Pichardo, James Fry, SaWo, Stephen C
21	Javaエージェント	Display Name, mnoronha
22	Javaコードの	Blubberguy22, Burkhard, Caleb Brinkman, Carter Brainerd, Community, Do Nhu Vy, Emil Sierżęga, George Bailey, Gerald Mücke, hd84335, ipsi, Kevin Thorne, Martijn Woudstra, Mitch Talmadge, Nagesh Lakinepally, PizzaFrog, Radouane ROUFID, RamenChef, sargue, Stephan, Stephen C, Trevor Sears, Universal Electricity
23	Javaコードの	Tony
24	Javaコマンド-'java'と'javaw'	4444, Ben, mnoronha, Stephen C, Vogel612
25	Javaコンパイラ-'javac'	CraftedCart, Jatin Balodhi, Mark Stewart, nishizawa23, Stephen C, Shadowfa, Tom Gijselinck
26	Javaソケット	Nikhil R
27	Javaでの2Dグラフィックス	17slim, ABDUL KHALIQ
28	Javaデプロイメント	garg10may, nishizawa23, Pseudonym Patel, RamenChef, Smit,

Stephen C		
29	Javaでのスクリプトをにする	Nikhil R
30	Javaネイティブアクセス	Ezekiel Baniaga, Stephan, Stephen C
31	Javaネイティブインタフェース	Coffee Ninja, Fjoni Yzeiri, Jorn Vernee, RamenChef, Stephen C, user1803551
32	JavaのJSON	Asaph, Bogdan Korinnyi, Burkhard, Cache Staheli, hd84335, ipsi, Jared Hooper, Kurzalead, MikaelF, Mrunal Pagnis, Nicholas J Panella, Nikita Kurtin, ppeterka, Prem Singh Bist, RamenChef, Ray Kiddy, SirKometa, still_learning, Stoyan Dekov, systemfreund, Tim, Vikas Gupta, vsminkov, Yuri Fedorov
33	Javaのとし - NullとNullPointerException	17slim, Andrii Abramov, Daniel Nugent, dorukayhan, fabian, François Cassin, Miles, Stephen C, Zircon
34	Javaのとし - スレッドと	dorukayhan, james large, Stephen C
35	Javaのとし - パフォーマンスの	Dorian, GPI, John Starich, Jorn Vernee, Michał Rybak, mnoronha, ppeterka, Sharon Rozinsky, steffen, Stephen C, xTrollxDudex
36	Javaのとし - の	Bhoomika, bruno, dimo414, Gal Dreiman, hd84335, SachinSarawgi, scorpp, Stephen C, Stephen Leppik, user3105453
37	Javaのとし - の	Alex T., Cody Gray, Enwired, Friederike, Gal Dreiman, hd84335, Hiren, Peter Rader, piyush_baderia, RamenChef, Ravindra HV, RudolphEst, Stephen C, Todd Sewell, user3105453
38	Javaパフォーマンスチューニング	Gene Marin, jatanp, Stephen C, Vogel612
39	Javaプラグインシステムの	Alexiy
40	Javaプリントサービス	Danilo Guimaraes, Leonardo Pina
41	Javaメモリモデル	Shree, Stephen C, Suminda Sirinath S. Dharmasena
42	Javaメモリ	Daniel M., engineerencoding, fgb, John Nash, jwd630, mnoronha, OverCoder, padippist, RamenChef, Squidward, Stephen C
43	JavaマシンJVM	Dushman, RamenChef, Rory McCrossan, Stephen Leppik
44	Java	Dariusz, hd84335, HTNW, Ilya, Mr. P, Petter Friberg, ravthiru, Stephen C, Stephen Leppik, Vogel612
45	JAXB	Dariusz, Drunix, fabian, hd84335, Jabir, ppeterka, Ram, Stephan, Thomas Fritsch, vallismortis, Walery Strauch
46	JAXP APIをしたXML	GPI
47	JAX-WS	ext1812, Jonathan Barbero, Stephen Leppik
48	JMX	esin88
49	JNDI	EJP, neohope, RamenChef

50	JShell	ostrichofevil , Sudip Bhandari
51	JVMツールインタフェース	desilijic
52	JVMフラグ	Configure , RamenChef
53	log4j / log4j2	Daniel Wild , Fildor , HCarrasko , hd84335 , Mrunal Pagnis , Rens van der Heijden
54	MultiThreadedアプリケーションでThreadPoolExecutorを初期化する	Brendon Dugan
55	Nashorn JavaScriptエンジン	ben75 , ekaerovets , Francesco Menzani , hd84335 , Ilya , InitializeSahib , kasperijs , VatsalSura
56	NIO - ネットワーク	Matthieu , mnoronha
57	NumberFormat	arpit pandey , John Nash , RamenChef , ΦXocę Π Πεπεύα Ψ
58	RSA暗号	Dennis Kriechel , Drunix , iqbal_cs , Maarten Bodewes , Nicktar , Shog9
59	ServiceLoader	fabian , Florian Genser , Gerald Mücke
60	SortedMap	Amit Gujarathi
61	Stack-Walking API	manouti
62	staticキーワードの初期化	17slim , Amir Rachum , Andrew Brooke , Arthur , ben75 , CarManuel , Daniel Nugent , EJP , Hi I'm Frogatto , Mark Yisri , Sadiq Ali , Skepter , Squidward
63	StringBuffer	Amit Gujarathi
64	StringBuilder	Andrii Abramov , Cache Staheli , David Soroko , Enigo , fabian , fgb , JudgingNotJudging , KIRAN KUMAR MATAM , Nicktar , P.J.Meisch , Stephen C
65	sun.misc.Unsafe	4444 , Daniel Nugent , Grexis , Stephen C , Suminda Sirinath S. Dharmasena
66	ThreadLocal	Dariusz , Liju Thomas , Manish Kothari , Nithanim , taer
67	TreeMapとTreeSet	Malt , Stephen C
68	XJC	Danilo Guimaraes , fabian
69	XML XPath関数	17slim , manouti
70	XOM - XMLオブジェクトモデル	Arthur , Makoto
71	アサーション	Jonathan , Makoto , rajah9 , RamenChef , The Guy with The Hat , Uri Agassi
72	アプレット	ArcticLord , Enigo , MadProgrammer , ppeterka
73	イテレータとイテレータ	Abubakkar , Comic Sans , Dariusz , Hulk , Lukas Knuth , RamenChef , Stephen C , user1121883 , WillShackleford

74	インターフェイス	100rabh, A Boschman, Abhishek Jain, Adowrath, Alex Shesterov, Andrew Tobilko, Andrii Abramov, Cà phê đen, Chirag Parmar, Conrad.Dean, Daniel Käfer, devguy, DVarga, Hilikus, inovaovao, intboolstring, James Oswald, Jan Vladimir Mostert, JavaHopper, Johannes, Jojodmo, Jonathan, Jorn Vernee, Kai, kstandell, Laurel, Marvin, MikeW, Paul Nelson Baker, Peter Rader, ppovoski, Prateek Agarwal, Radouane ROUFID, RamenChef, Robin, Simulant, someoneigna, Stephen C, Stephen Leppik, Sujith Niraikulathan, Thomas Gerot, user187470, Vasiliy Vlasov, Vince Emigh, xwoker, Zircon
75	オーディオ	Dac Saunders, Petter Friberg, RamenChef, TNT, tonirush, Tot Zam, Vogel612
76	オートボクシング	17slim, Anony-Mousse, Bob Rivers, Chuck Daniels, cshubhamrao, fabian, hd84335, J Atkin, janos, kaartic, Kirill Sokolov, Luan Nico, Nayuki, piyush_baderia, Ram, RamenChef, Saagar Jha, Stephen C, Unihedron, Vladimir Vagaytsev
77	オブジェクトクラスのメソッドとコンストラクタ	A Boschman, Ad Infinitum, Andrii Abramov, Ani Menon, anuvab1911, Arthur Nosedá, augray, Brett Kail, Burkhard, CaffeineToCode, Chris Midgley, cricket_007, Dariusz, Elazar, Emil Sierżęga, Enigo, fabian, fgb, Floern, fzzfzzfzz, hd84335, intboolstring, james large, JamesENL, Jens Schauder, John Slegers, Jorn Vernee, kstandell, Lahiru Ashan, Laurel, Miljen Mikic, mnoronha, mykey, NageN, Nayuki, Nicktar, Pace, Petter Friberg, Radouane ROUFID, Ram, Robert Columbia, Ronnie Wang, shmosel, Stephen C, TNT
78	オブジェクトのクローニング	Ayush Bansal, Christophe Weis, Jonathan
79	オブジェクト	Andrii Abramov, arcy, Vasiliy Vlasov
80	オプション	A Boschman, Abubakkar, Andrey Rubtsov, Andrii Abramov, assylas, bowmore, Charlie H, Chris H., Christophe Weis, compuhosny, Dair, Emil Sierżęga, enrico.bacis, fikovnik, Grzegorz Górkiiewicz, gwintrob, Hadson, hd84335, hzp, J Atkin, Jean-François Savard, John Slegers, Jude Niroshan, Maroun Maroun, Michael Wiles, OldMcDonald, shmosel, Squidward, Stefan Dollase, Stephen C, ultimate_guy, Unihedron, user140547, Vince, vsminkov, xwoker
81	オラクルのコード	Ahmed Ashour, aioobe, akhils, alex s, Andrii Abramov, Cassio Mazzochi Molin, Dan Whitehouse, Enigo, erickson, f_puras, fabian, giucal, hd84335, J.D. Sandifer, Lahiru Ashan, Mac70, NamshubWriter, Nicktar, Petter Friberg, Pradatta, Pritam Banerjee, RamenChef, sanjaykumar81, Santa Claus, Santhosh Ramanan, VGR
82	カプセル	Adam Ratzman, Adil, Daniel M., Drayke, VISHWANATH N P
83	カレンダーとそのサブクラス	Bob Rivers, cdm, kann, Makoto, mnoronha, ppeterka, Ram, VGR
84	クラス - Java Reflection	gobes, KIRAN KUMAR MATAM
85	クラスとオブジェクト	Community, Configure, Daniel LIn, Dave Ranjan, EJP, eveysky, fabian, Jens Schauder, Kevin Johnson, KIRAN KUMAR MATAM, MasterBlaster, Mureinik, Rakitić, Ram, RamenChef, Ryan Cocuzzo, Salman Kazmi, Tyler Zika
86	クラスパス	Aaron Digulla, GPI, K'', Kenster, Ruslan Ulanov, Stephen C,

trashgod		
87	クラスローダー	FFY00, Flow, Holger, Makoto, Stephen C
88	ゲッターとセッター	Fildor, Ironcache, Kröw, martin, Petter Friberg, Stephen C, Sujith Niraikulathan, Thisaru Guruge, uzaif
89	コマンドラインの	Burkhard, Michael von Wenckstern, Stephen C
90	コレクション	4castle, A_Arnold, Ad Infinitum, Alek Mieczkowski, alex s, altomnr, Andy Thomas, Anony-Mousse, Ashok Felix, Aurasphere, Bob Rivers, ced-b, ChandrasekarG, Chirag Parmar, clinomaniac, Codebender, Craig Gidney, Daniel Stradowski, dcod, DimaSan, Dušan Rychnovský, Enigo, Eran, fabian, fgb, GPI, Grzegorz Górkiwicz, ionyx, Jabir, Jan Vladimir Mostert, KartikKannapur, Kenster, KIRAN KUMAR MATAM, koder23, KudzieChase, Makoto, Maroun Maroun, Martin Frank, Matseman, Mike H, Mo.Ashfaq, Mrunal Pagnis, mystarocks, Oleg Sklyar, Pablo, Paweł Albecki, Petter Friberg, philnate, Polostor, Poonam, Powerlord, ppeterka, Prasad Reddy, Radiodef, rajadilipkolli, rd22, rdonuk, Ruslan Bes, Samk, SJB, Squidward, Stephen C, Stephen Leppik, Unihedron, user2296600, user3105453, Vasiliy Vlasov, Vasily Kabunov, VatsalSura, vsminkov, webo80, xploreraaj
91	コレクションの	John DiFinì
92	コレクションファクトリメソッド	Jacob G.
93	コンカレントコレクション	GPI, Kenster, Powerlord, user2296600
94	コンストラクタ	Andrii Abramov, Asiat, BrunoDM, ced-b, Codebender, Dylan, George Bailey, Jeremy, Ralf Kleberhoff, RamenChef, Thomas Gerot, tynn, Vogel612
95	コンソール I / O	Aaron Franke, Ani Menon, Erkan Haspulat, Francesco Menzani, jayantS, Lankymart, Loris Securo, manetsus, Olivier Grégoire, Petter Friberg, rove, Saagar Jha, Stephen C
96	ジェネリックス	1d0m3n30, 4444, Aaron Digulla, Abhishek Jain, Alex Meiburg, alex s, Andrei Maieras, Andrii Abramov, Anony-Mousse, Bart Enkelaar, bitek, Blubberguy22, Bob Brinks, Burkhard, Cache Staheli, Cannon, Ce7, Chriss, codell, Codebender, Daniel Figueroa, daphshez, DVarga, Emil Sierżęga, enrico.bacis, Eran, faraa, hd84335, hexafraction, Jan Vladimir Mostert, Jens Schauder, Jorn Vernee, Jude Niroshan, kcopppock, Kevin Montrose, Lahiru Ashan, Lii, manfcas, Mani Muthusamy, Marc, Matt, Mistalis, Mshnik, mvd, Mzzzzzz, NatNgs, nishizawa23, Oleg Sklyar, Onur, Ortomala Lokni, paisanco, Paul Bellora, Paweł Albecki, PcAF, Petter Friberg, phant0m, philnate, Radouane ROUFID, RamenChef, rap-2-h, rd22, Rogério, rove, RutledgePaulV, S.K. Venkat, Siguza, Stephen C, Stephen Leppik, suj1th, tainy, ThePhantomGamer, Thomas, TNT, 7oleæz æq7 qoq, Unihedron, Vlad-HC, Wesley, Wilson, yiwei, Yuriy Fedorov
97	ジャストインタイム JIT コンパイラ	Liju Thomas, Stephen C
98	シリアライゼーション	akhilsk, Batty, Bilesh Ganguly, Burkhard, EJP, emotionlessbananas, faraa, GradAsso, KIRAN KUMAR MATAM, noscreenname, Onur, rokonoid, Siva Sainath Reddy Bandi, Vasilis Vasilatos, Vasiliy Vlasov

99	シングルトン	aasu , Andrew Antipov , Daniel Käfer , Dave Ranjan , David Soroko , Emil Sierżęga , Enigo , fabian , Filip Smola , GreenGiant , Gubbel , Hulk , Jabir , Jens Schauder , JonasCz , Jonathan , JonK , Malt , Matsemann , Michael Lloyd Lee mlk , Mifeet , Miroslav Bradic , NamshubWriter , Pablo , Peter Rader , RamenChef , riyaz-ali , sanastasiadis , shmosel , Stefan Dollase , stefanobaghino , Stephen C , Stephen Leppik , still_learning , Uri Agassi , user3105453 , Vasiliy Vlasov , Vlad-HC , Vogel612 , xploreraJ
100	スーパーキーワード	Abhijeet
101	スキャナ	Alek Mieczkowski , Chirag Parmar , Community , Jon Ericson , JonasCz , Ram , RamenChef , Redterd , Stephen C , sun-solar-arrow , Xocę Π epeúpa ツ
102	ストリーム	4castle , Abubakkar , acdcjunior , Aimee Borda , Akshit Soota , Amitay Stern , Andrew Tobilko , Andrii Abramov , ArsenArsen , Bart Kummel , berko , Blubberguy22 , bpoiss , Brendan B , Burkhard , Cerbrus , Charlie H , Claudio , Community , Conrad.Dean , Constantine , Daniel Käfer , Daniel M. , Daniel Stradowski , Dariusz , David G. , DonyorM , Dth , Durgpal Singh , Dushko Jovanovski , DVarga , dwursteisen , Eirik Lygre , enrico.bacis , Eran , explv , Fildor , Gal Dreiman , gontard , GreenGiant , Grzegorz Oledzki , Hank D , Hulk , iliketocode , ItachiUchiha , izikovic , J Atkin , Jamie Rees , JavaHopper , Jean-François Savard , John Slegers , Jon Erickson , Jonathan , Jorn Vernee , Jude Niroshan , JudgingNotJudging , Justin , Kapep , Kip , LisaMM , Makoto , Malt , malteo , Marc , MasterBlaster , Matt , Matt , Matt S. , Matthieu , Michael Piefel , MikeW , Mitch Talmadge , Mureinik , Muto , Naresh Kumar , Nathaniel Ford , Nuri Tasdemir , OldMcDonald , Oleg L. , omiel , Ortomala Lokni , Pawan , Paweł Albecki , Petter Friberg , Philipp Wendler , philnate , Pirate_Jack , ppeterka , Radnyx , Radouane ROUFID , Rajesh Kumar , Rakitić , RamenChef , Ranadip Dutta , ravthiru , reto , Reut Sharabani , RobAu , Robin , Roland Illig , Ronnie Wang , rrampage , RudolphEst , sargue , Sergii Bishyr , sevenforce , Shailesh Kumar Dayananda , shmosel , Shoe , solidcell , Spina , Squidward , SRJ , stackptr , stark , Stefan Dollase , Stephen C , Stephen Leppik , Steve K , Sugan , sujlth , thiagogcm , tpunt , Tunaki , Unihedron , user1133275 , user1803551 , Valentino , vincentvanjoe , vsnyc , Wilson , Ze Rubeus , zwl
103	セキュリティ	John Nash , shibli049
104	セット	A_Arnold , atom , ced-b , Chirag Parmar , Daniel Stradowski , demongolem , DimaSan , fabian , Kaushal28 , Kenster
105	ソケット	Ordriel
106	タイプ	4castle , Filip Smola , Joshua Carmody , Nick Donnelly , RamenChef , Squidward
107	デキューインタフェース	Suketu Patel
108	デフォルトメソッド	ar4ers , hd84335 , intboolstring , javac , Jeffrey Bosboom , Jens Schauder , Kai , matt freake , o_nix , philnate , Ravindra HV , richersoon , Ruslan Bes , Stephen C , Stephen Leppik , Vasiliy Vlasov
109	ネットワーキング	Arthur , Burkhard , devnull69 , DonyorM , glee8e , Grayson Croom , Ilya , Malt , Matej Kormuth , Matthieu , Mine_Stone , ppeterka , RamenChef , Stephen C , Tot Zam , vsav

110	バイトコードの	bloo , Display Name , rakwaht , Squidward
111	パッケージ	JamesENL , KIRAN KUMAR MATAM
112	ハッシュ	KIRAN KUMAR MATAM
113	ビット	Aimee Borda , Blubberguy22 , dosdebug , esin88 , Gerald Mücke , Jorn Vernee , Kineolyan , mnoronha , Nayuki , Rednivrug , Ryan Hilbert , Stephen C , thatguy
114	ファイル / O	Alper Firat Kaya , Arthur , assylas , ata , Aurasphere , Burkhard , Conrad.Dean , Daniel M. , Enigo , FlyingPiMonster , Gerald Mücke , Gubbel , Hay , hd84335 , Jabir , James Jensen , Jason Sturges , Jordy Baylac , leaqui , mateuscb , MikaelF , Moshiour , Myridium , Nicktar , Peter Gordon , Petter Friberg , ppeterka , RAnders00 , RobAu , rokonoid , Sampada , sebkur , ShivBuyya , Squidward , Stephen C , still_learning , Tilo , Tobias Friedinger , TuringTux , Will Hardwick-Smith
115	フォーク / フレームワークによるプログラミング	Community , Joe C
116	プリミティブデータ	17slim , 1d0m3n30 , Amani Kilumanga , Ani Menon , Anony-Mousse , Bilesh Ganguly , Bob Rivers , Burkhard , Conrad.Dean , Daniel , Dariusz , DimaSan , dnup1092 , Do Nhu Vy , enrico.bacis , fabian , Francesco Menzani , Francisco Guimaraes , gar , Ilya , IncrediApp , ipsi , J Atkin , JakeD , javac , Jean-François Savard , Jojodmo , Kapep , KdgDev , Lahiru Ashan , Master Azazel , Matt , mayojava , MBorsch , nimrod , Pang , Panther , ParkerHalo , Petter Friberg , Radek Postolowicz , Radouane ROUFID , RAnders00 , RobAu , Robert Columbia , Simulant , Squidward , Stephen C , Stephen Leppik , Sundeeep , SuperStormer , ThePhantomGamer , TMN , user1803551 , user2314737 , Veedrac , Vogel612
117	プログラムによるの	alain.janinm , Dariusz , kajacx , Kenster , mnoronha
118	プロセス	Andy Thomas , Bob Rivers , ppeterka , vorburger , yitzih
119	プロパティクラス	17slim , Arthur , J Atkin , Jabir , KIRAN KUMAR MATAM , Marvin , peterh , Stephen C , VGR , vorburger
120	ベンチマーク	esin88
121	モジュール	Jonathan , user140547
122	ユニットテスト	Ironcache
123	ラムダ	Abhishek Jain , Ad Infinitum , Adam , aioobe , Amit Gupta , Andrei Maieras , Andrew Tobilko , Andrii Abramov , Ankit Katiyar , Anony-Mousse , assylas , Brian Goetz , Burkhard , Conrad.Dean , cringe , Daniel M. , David Soroko , dimitrisli , Draken , DVarga , Emre Bolat , enrico.bacis , fabian , fgb , Gal Dreiman , gar , GPI , Hank D , hexafraction , Ivan Vergiliev , J Atkin , Jean-François Savard , Jeroen Vandevelde , John Slegers , JonasCz , Jorn Vernee , Jude Niroshan , JudgingNotJudging , Kevin Raoofi , Malt , Mark Green , Matt , Matthew Trout , Matthias Braun , ncmathsadist , nobeh , Ortomala Lokni , Paūlo Ebermann , Paweł Albecki , Petter Friberg , philnate , Pujan Srivastava , Radouane ROUFID , RamenChef , rolve , Saclyr Barlonium , Sergii Bishyr , Skylar Sutton , solomonope , Stephen C , Stephen Leppik , timbooo , Tunaki , Unihedron , vincentvanjoe , Vlasec , Vogel612 , webo80 , William Ritson ,

		Wolfgang, Xaerxess, xploreraj, Yogi, Ze Rubeus
124	リスト	17slim, A Boschman, Arthur, Avinash Kumar Yadav, Blubberguy22, ced-b, Daniel Nugent, granmirupa, Ilya, Jan Vladimir Mostert, janos, JD9999, jopasserat, Karthikeyan Vaithilingam, Kenster, Krzysztof Krason, Oleg Sklyar, RamenChef, Sheshnath, Stephen C, sudo, Thisaru Guruge, Vasilis Vasilatos, ΦXocę Π Πεπεúpa ツ
125	リスト→SET	KIRAN KUMAR MATAM
126	リソース→クラスパス→	Androbin, Christian, Emily Mabrey, Enwired, fabian, Gerald Mücke, Jesse van Bekkum, Kenster, Stephen C, timbooo, VGR, vorburger
127	リテラル	1d0m3n30, EJP, ParkerHalo, Stephen C, ThePhantomGamer
128	リフレクションAPI	Ali786, ArcticLord, Aurasphere, Blubberguy22, Bohemian, Christophe Weis, Drizzt321, fabian, hd84335, Joeri Hendrickx, Luan Nico, madx, Michael Myers, Onur, Petter Friberg, RamenChef, Ravindra babu, Squidward, Stephen C, Tony BenBrahim, Universal Electricity, ΦXocę Π Πεπεúpa ツ
129	リモートメソッド→呼び出し→RMI→	RamenChef, smichel, Stephen C, user1803551, Vasiliy Vlasov
130	リンクされたハッシュマップ	Amit Gujarathi, KIRAN KUMAR MATAM
131	ローカリゼーションと→	Code.IT, dimo414, Eduard Wirch, emotionlessbananas, Squidward, sun-solar-arrow
132	ローカルインナークラス	KIRAN KUMAR MATAM
133	ロギング→java.util.logging→	bn., Christophe Weis, Emil Sierżęga, P.J.Meisch, vallismortis
134	→→→なJavaの→とし→	akvyalkov, Anand Vaidya, Andy Thomas, Anton Hlinisty, anuvab1911, Conrad.Dean, Daniel Nugent, Dushko Jovanovski, Enwired, Gal Dreiman, Gerald Mücke, HTNW, james large, Jenny T-Type, John Starich, Lahiru Ashan, Makoto, Morgan Zhang, NamshubWriter, P.J.Meisch, Pirate_Jack, ppeterka, RamenChef, screab, Siva Sankar Rajendran, Squidward, Stephen C, Stephen Leppik, Steve Harris, tonirush, TuringTux, user3105453
135	→オブジェクト	1d0m3n30, Bohemian, Holger, Idcmp, Jon Ericson, kristyna, Michael Piefel, Stephen C, Vogel612
136	→クラス	Mykola Yashchenko
137	→プログラミング→スレッド→	adino, Alex, assylia, bfd, Bhagyashree Jog, bowmore, Burkhard, Chetya, corsiKa, Dariusz, Diane Chastain, DimaSan, dimo414, Fildor, Freddie Coleman, GPI, Grzegorz Górkiwicz, hd84335, hellrocker, hexafraction, Ilya, james large, Jens Schauder, Johannes, Jorn Vernee, Kakarot, Lance Clark, Malt, Matěj Kripner, Md. Nasir Uddin Bhuiyan, Michael Piefel, michaelbahr, Mitchell Tracy, MSB, Murat K., Mureinik, mvd, NatNgs, nickguletskii, Olivier Durin, OlivierTheOlive, Panda, parakmiakos, Paweł Albecki, ppeterka, RamenChef, Ravindra babu, rd22, RudolphEst, snowe2010, Squidward, Stephen C, Sudhir Singh, Tobias Friedinger, Unihedron, Vasiliy Vlasov, Vlad-HC, Vogel612, wolfcastle, xTrollxDudex, YCF_L, Yury Fedorov, ZX9
138	→→→→	Arthur, David Grant, David Soroko, dorukayhan, F. Stephen Q,

		Kichiin, MasterBlaster, michaelbahr, rokonoid, Stephen C, Thodgnir
139	コレクション	mnoronha, ppeterka, Viacheslav Vedenin
140	と	Adrian Krebs, agilob, akhilsk, Andrii Abramov, Bhavik Patel, Burkhard, Cache Staheli, Codebender, Dariusz, DarkV1, dimo414, Draken, EAX, Emil Sierżęga, enrico.bacis, fabian, FMC, Gal Dreiman, GreenGiant, Hernanibus, hexafraction, Ilya, intboolstring, Jabir, James Jensen, JavaHopper, Jens Schauder, John Nash, John Slegers, JonasCz, Kai, Kevin Thorne, Malt, Manish Kothari, Md. Nasir Uddin Bhuiyan, michaelbahr, Miljen Mikic, Mitch Talmadge, Mrunal Pagnis, Myridium, mzc, Nikita Kurtin, Oleg Sklyar, P.J.Meisch, Paweł Albecki, Peter Gordon, Petter Friberg, ppeterka, Radek Postołowicz, Radouane ROUFID, Raj, RamenChef, rdonuk, Renukaradhya, RobAu, sandbo00, Saša Šijak, sharif.io, Stephen C, Stephen Leppik, still_learning, Sudhir Singh, sv3k, tatoalo, Thomas Fritsch, Tripta Kiroula, vic-3, Vogel612, Wilson, yiwei
141		alphaloop, hexafraction, Uux
142	のクラスとクラス	ChemicalFlash, DimaSan, fgb, hd84335, Mshnik, RamenChef, Sandesh, sargue, Slava Babin, Stephen C, tynn
143		Andy Thomas, atom, Bobas_Pett, Ce7, charlesreid1, Configure, David Soroko, fabian, hamena314, hd84335, JavaHopper, Javant, Matej Kormuth, mayojava, Nicktar, Peter Gordon, RamenChef, Raviteja, Ruslan Bes, Stephen C, sumit
144		1d0m3n30, A Boschman, aioobe, Amani Kilumanga, Andreas Fester, Andrew Sklyarevsky, Andrew Tobilko, Andrii Abramov, Anony-Mousse, bcosynot, Bob Rivers, coder-croc, Community, Constantine, Daniel Käfer, Daniel M., Danilo Guimaraes, DVarga, Emil Sierżęga, enrico.bacis, f_puras, fabian, Gal Dreiman, Gene Marin, Grexis, Grzegorz Oledzki, ipsi, J Atkin, Jared Hooper, javac, Jérémie Bolduc, Johannes, Jon Ericson, k3b, Kenster, Lahiru Ashan, Maarten Bodewes, madx, Mark, Michael Myers, Mick Mnemonic, NageN, Nefl0, Nolequen, OldCurmudgeon, OliPro007, OverCoder, P.J.Meisch, Panther, Paweł Albecki, Petter Friberg, Punika, Radouane ROUFID, RamenChef, rd22, Ronon Dex, Ryan Hilbert, S.K. Venkat, Samk, shmosel, Spina, Stephen Leppik, Tarun Maganti, Tim, Torsten, VGR, Victor G., Vinay, Wolf, Yury Fedorov, Zefick, Xocę, Xepeúpa
145	メソッドディスパッチ	Jeet
146	タイプ	Daniel Nugent, Stephen C, Suminda Sirinath S. Dharmasena, xTrollxDudex
147	データ	Do Nhu Vy, giucal, Jorn Vernee, Lord Farquaad, Yohanes Khosiawan
148		EJP, NageN, Thisaru Guruge
149	クラスのメンバーへのアクセスをする	Aasmund Eldhuset, Abhishek Balaji R, Catalina Island, Daniel M., intboolstring, Jonathan, Mark Yisri, Mureinik, NageN, ParkerHalo, Stephen C, Vogel612
150		17slim, agilob, alain.janinm, ata, Binary Nerd, Burkhard, coobird, Dmitriy Kotov, Durgpal Singh, Emil Sierżęga, Emily Mabrey, Enigo, fabian, GPI, hd84335, J Atkin, Jabir, Javant,

		Javier Diaz, Jeffrey Bosboom, johnnyaug, Jonathan, Kakarot, KartikKannapur, Kenster, michaelbahr, Mo.Ashfaq, Nathaniel Ford, phatfingers, Ram, RamenChef, ravthiru, sebkur, Stephen C, Stephen Leppik, Viacheslav Vedenin, VISHWANATH N P, Vogel612
151	□□□なコントロール□□	Adrian Krebs, AJNeufeld, Andrew Brooke, AshanPerera, Buddy, Caleb Brinkman, Cas Eliëns, Coffeehouse Coder, CraftedCart, dedmass, ebo, fabian, intboolstring, Inzimam Tariq IT, Jens Schauder, JonasCz, Jorn Vernee, juergen d, Makoto, Matt Champion, philnate, Ram, Santhosh Ramanan, sevenforce, Stephen C, teek, Unihedron, Uri Agassi, xworker
152	□□□□□□	Daniel Nugent, Dushman, Omar Ayala, Rafael Pacheco, RamenChef, VGR, xsami
153	□□	Adrian Krebs, Amani Kilumanga, Daniel LIn, Dushman, Kakarot, Lernkurve, Markus L, NageN, Pawan, Ravindra babu, Saiful Azad, Stephen C
154	□□なオブジェクト	Ankit Katiyar
155	□□□□な□□	Adowrath, Kishore Tulsiani, WillShackleford
156	□□□コマンド	RamenChef
157	□	1d0m3n30, Andreas, EJP, Li357, RamenChef, shmosel, Stephen C, Stephen Leppik
158	□□いハッシュマップ	Amit Gujarathi, KIRAN KUMAR MATAM
159	□ち□□と□ち□□	Ad Infinitum, Alek Mieczkowski, Androbin, DimaSan, engineercoding, ppeterka, RamenChef, rd22, Samk, Stephen C
160	□□で□まる□□□	Sugan
161	□□コード	Ilya
162	□□□	17slim, A.J. Brown, A_Arnold, Abhishek Jain, Abubakkar, Adam Ratzman, Adrian Krebs, agilob, Aiden Deom, Alex Meiburg, Alex Shestеров, altomnr, Amani Kilumanga, Andrew Tobilko, Andrii Abramov, Andy Thomas, Anony-Mousse, Asaph, Ataeraxia, Austin, Austin Day, ben75, bfd, Bob Brinks, bpoiss, Burkhard, Cache Staheli, Caner Balım, Chris Midgley, Christian, Christophe Weis, coder-croc, Community, cyberscientist, Daniel Käfer, Daniel Stradowski, DarkV1, dedmass, DeepCoder, dnup1092, dorukayhan, drov, DVarga, ekeith, Emil Sierżęga, emotionlessbananas, enrico.bacis, Enwired, fabian, FlyingPiMonster, Gabriele Mariotti, Gal Dreiman, Gergely Toth, Gihan Chathuranga, GingerHead, giucal, Gray, GreenGiant, hamena314, Harish Gyanani, HON95, iliketocode, Ilya, Infuzed guy, intboolstring, J Atkin, Jabir, javac, JavaHopper, Jeffrey Lin, Jens Schauder, Jérémie Bolduc, John Slegers, Jojodmo, Jon Ericson, JonasCz, Jordi Castilla, Jorn Vernee, JSON C11, Jude Niroshan, Kamil Akhuseyinoglu, Kapep, Kaushal28, Kaushik NP, Kehinde Adedamola Shittu, Kenster, kstandell, Lachlan Dowding, Lahiru Ashan, Laurel, Leo Aso, Liju Thomas, LisaMM, M.Sianaki, Maarten Bodewes, Makoto, Malav, Malt, Manoj, Manuel Spigolon, Mark Stewart, Marvin, Matej Kormuth, Matt Clark, Matthias Braun, maxdev, Maxim Plevako, mayha, Michael, MikeW, Miles, Miljen Mikic, Misa Lazovic, mr5, Myridium, NikolaB, Nufail, Nuri Tasdemir, OldMcDonald, OliPro007, Onur, Optimiser, ozOli, P.J.Meisch, Paolo Forgia, Paweł Albecki, Petter Friberg, phant0m,

		piyush_baderia, ppeterka, Přemysl Šťastný, PSo, QoP, Radouane ROUFID, Raj, RamenChef, RAnders00, Rocherlee, Ronnie Wang, Ryan Hilbert, ryanyuyu, Sayakiss, SeeuD1, sevenforce, Shaan, ShivBuyya, Shoe, Sky, SmS, solidcell, Squidward, Stefan Isele - prefabware.com, stefanobaghino, Stephen C, Stephen Leppik, Steven Benitez, still_learning, Sudhir Singh, Swanand Pathak, S hadomfa, TDG, TheLostMind, ThePhantomGamer, Tony BenBrahim, Unihedron, VGR, Vishal Biyani, Vogel612, vsminkov, vvtx, Wilson, winseybash, xworker, yuku, Yury Fedorov, Zachary David Saunders, Zack Teater, Ze Rubeus, ΦXocę Π Pepeúpa ツ
163	トークナイザ	M M
164	との	Chirag Parmar, DarkV1, Gihan Chathuranga, Jabir, JonasCz, Kaushal28, Lachlan Dowding, Laurel, Maarten Bodewes, Matt Clark, PSo, RamenChef, Shaan, Stephen C, still_learning
165	をにする	Bohemian
166	しいファイル / O	dorukayhan, niheno, TuringTux
167	クラス	A_Arnold, alain.janinm, arcy, Bob Rivers, Christian Wilkie, explv, Jabir, Jean-Baptiste Yunès, John Smith, Matt Clark, Miles, NamshubWriter, Nicktar, Nishant123, Ph0bi4, ppeterka, Ralf Kleberhoff, Ram, skia.heliou, Squidward, Stephen C, Vinod Kumar Kashyap
168	と java.time.*	Bilbo Baggins, bowmore, Michael Piefel, Miles, mnoronha, Simon, Squidward, Tarun Maganti, Vogel612, ΦXocę Π Pepeúpa ツ
169	インタフェース	Andreas
170		Amani Kilumanga, Andy Thomas, Asaph, ced-b, Daniel M., fabian, hd84335, intboolstring, kaotikmynd, Laurel, Makoto, nhahtdh, ppeterka, Ram, RamenChef, Saif, Tot Zam, Unihedron, Vogel612
171	なコンパレータ	Andrii Abramov, Conrad.Dean, Daniel Nugent, fabian, GPI, Hazem Farahat, JAVAC, Mshnik, Nolequen, Petter Friberg, Prateek Agarwal, sebkur, Stephen C
172		Ad Infinitum, Alon .G., Andrei Maieras, Andrii Abramov, bruno, Conrad.Dean, Dariusz, Demon Coldmist, Drizzt321, Dushko Jovanovski, fabian, faraa, GhostCat, hd84335, Hendrik Ebbbers, J Atkin, Jorn Vernee, Kapep, Malt, MasterBlaster, matt freake, Nolequen, Ortomala Lokni, Ram, shmosel, Stephen C, Umberto Raimondi, Vogel612, ΦXocę Π Pepeúpa ツ
173	なインターフェイス	bn., noscreenname, P.J.Meisch, RamenChef, TuringTux
174		17slim, 1d0m3n30, A Boschman, acdcjunior, afuc func, AJ Jwair, Amani Kilumanga, Andreas, Andrew, Andrii Abramov, Blake Yarbrough, Blubberguy22, Bobas_Pett, c.uent, Cache Staheli, Chris Midgley, Claudia, clinomaniac, Dariusz, Darth Shadow, Davis, EJP, Emil Sierżęga, Eran, fabian, FedeWar, FlyingPiMonster, futureelite7, Harsh Vakharia, hd84335, J Atkin, JavaHopper, Jérémie Bolduc, jimrm, Jojodmo, Jorn Vernee, kanhaiya agarwal, Kevin Thorne, Li357, Loris Securo, Lynx Brutal, Maarten Bodewes, Mac70, Makoto, Marvin, Michael Anderson, Mshnik, NageN, Nuri Tasdemir, Ortomala Lokni, OverCoder, ParkerHalo, Peter Gordon, ppeterka, qxz, rahul tyagi, RamenChef, Ravan, Reut Sharabani, Rubén, sargue, Sean Owen, ShivBuyya, shmosel, SnoringFrog, Stephen C, tonirush, user3105453, Vogel612

, Winter		
175	□□□□	100rabh, A_Arnold, Alex, Andrii Abramov, Bob Rivers, Cache Staheli, DimaSan, Jasper, Kakarot, Kuroda, Manuel Vieda, Michael Piefel, phatfingers, RamenChef, Skylar Sutton, Vivek Anoop
176	□□な□□のための AppDynamicsとTIBCO BusinessWorks□□	Alexandre Grimaud
177	□□	Ad Infinitum, Adam, Adrian Krebs, agoeb, Ali Dehghani, Andrii Abramov, ar4ers, Arkadiy, Blubberguy22, Bohemian, Brad Larson, Burkhard, CodeCore, coder-croc, Dariusz, David Grinberg, devnull69, DonyorM, DVarga, Emre Bolat, explv, fabian, gattsbr, geniushkg, GhostCat, Gubbel, hirosh, HON95, J Atkin, Jason V, JavaHopper, Jeffrey Bosboom, Jens Schauder, Jonathan, Jorn Vernee, Kai, Kevin DiTraglia, kiuby_88, Lahiru Ashan, Luan Nico, maheshkumar, Mshnik, Muhammed Refaat, OldMcDonald, Oleg Sklyar, Ortomala Lokni, PM 77-1, Prateek Agarwal, QoP, Radouane ROUFID, RamenChef, Ravindra babu, Shog9, Simulant, SJB, Slava Babin, Stephen C, Stephen Leppik, still_learning, Sudhir Singh, Theo, ToTheMaximum, uhrm, Unihedron, Vasiliy Vlasov, Vucko
178	□□リリースのJARファイル	manouti
179	□□	RAnders00
180	□□と□□	JD9999, KIRAN KUMAR MATAM, Mureinik, Stephen C, VatsalSura
181	□アセンブルと□コンパイル	ipsi, mnoronha
182	□□とマネー	Alexey Lagunov
183	□□	3442, 416E64726577, A Boschman, A.M.K, A_Arnold, Abhishek Jain, Abubakkar, acdcjunior, Ad Infinitum, Addis, Adrian Krebs, AER, afzalex, agilob, Alan, Alex Shesterov, Alexandru, altomnr, Amani Kilumanga, Andrew Tobilko, Andrii Abramov, AndroidMechanic, Anil, ankidaemon, ankit dassor, anotherGatsby, antonio, Ares, Arthur, Ashish Ahuja, assylas, AstroCB, baao, Beggs, Berzerk, Big Fan, BitNinja, bjb568, Blubberguy22, Bob Rivers, bpoiss, Bryan, BudsNanKis, Burkhard, bwegs, clphr, Cache Staheli, Cerbrus, Charitha, Charlie H, Chris Midgley, Christophe Weis, Christopher Schneider, Codebender, coder-croc, Cold Fire, Colin Pickard, Community, Configure, CptEric, Daniel Käfer, Daniel Stradowski, Dariusz, DarkV1, David G., DeepCoder, Devid Farinelli, Dhrubajyoti Gogoi, Dmitry Ginzburg, dorukayhan, Duh-Wayne-101, Durgpal Singh, DVarga, Ed Cottrell, Edwin Tenovimas, Eilit, eisbehr, Elad, Emil Sierżęga, Emre Bolat, Eng.Fouad, enrico.bacis, Eran, Erik Minarini, Etki, explv, fabian, fedorqui, Filip Haglund, Forest White, fracz, Franck Dernoncourt, Functino, futureelite7, Gal Dreiman, gar, Gene Marin, GingerHead, granmirupa, Grexis, Grzegorz Sancewicz, Gubbel, Guilherme Torres Castro, Gustavo Coelho, hhj8i, Hiren, Idos, ihatecsv, iliketocode, Ilya, Ilyas Mimouni, intboolstring, Irfan, J Atkin, jabbathehutt1234, JakeD, James Taylor, Jamie, Jamie Rees, Janez Kuhar, Jared Rummler, Jargonius, Jason Sturges, JavaHopper, Javant, Jeeter, Jeffrey Bosboom, Jens Schauder, Jérémie Bolduc, Jeutnarg, jhnance, Jim Garrison, jitendra varshney, jmattheis, Joffrey, Johannes, johannes_preiser, John Slegers, JohnB, Jojodmo, Jonathan, Jordi Castilla, Jorn, Jorn Vernee, Josh, JStef, JudgingNotJudging, Justin, Kapep, KartikKannapur,

