無料電子ブック

学習

Kotlin

Free unaffiliated eBook created from
Stack Overflow contributors.

#kotlin

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: kotlin

It is an unofficial and free Kotlin 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 Kotlin.

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: Kotlinをいめる

Kotlinは、にJVMをとするJetBrainsによってされたオブジェクトプログラミングです。 Kotlinは、コンパイル、、に、100Javaとのというをってされています。 Kotlinはまた、Javaがむくのをすることをにされています。 Kotlinのコンパイラでは、JVMのJavaバイトコードとJavaScriptのにコンパイルできます。

Kotlinをコンパイルする

Kotlinには、EclipseとIntelliJのIDEプラグインがあります。 KotlinはMavenをって、Antをって、Gradleをって 、またはコマンドラインからコンパイルすることもできます。

`$ kotlinc Main.kt`はjavaクラスファイルをします。この、 `MainKt.class` クラスにKtがされていることにしてください。しかし、 `$ java MainKt` javaをしてクラスファイルをすると、のがスローされます。

```
Exception in thread "main" java.lang.NoClassDefFoundError: kotlin/jvm/internal/Intrinsics
    at MainKt.main(Main.kt)
Caused by: java.lang.ClassNotFoundException: kotlin.jvm.internal.Intrinsics
    at java.net.URLClassLoader.findClass(URLClassLoader.java:381)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:335)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
    ... 1 more
```

Javaをしてのクラスファイルをするには、のクラスパスにKotlinランタイムJarファイルをめるがあります。

```
java -cp .:/path/to/kotlin/runtime/jar/kotlin-runtime.jar MainKt
```

バージョン

バージョン	
1.0.0	2016-02-15
1.0.1	2016-03-16
1.0.2	2016513
1.0.3	2016630
1.0.4	2016-09-22
1.0.5	2016-11-08

バージョン	
1.0.6	2016-12-27
1.1.0	2017-03-01
1.1.1	2017-03-14
1.1.2	2017-04-25
1.1.3	2017-06-23

Examples

こんにちは

てのKotlinプログラムは`main`からまります。なKotlinの "Hello World"プログラムのをにします。

```
package my.program

fun main(args: Array<String>) {
    println("Hello, world!")
}
```

のコードを`Main.kt`というのファイルに`Main.kt`ますこのファイルはにです

JVMをとする、ファンクションはファイルからしたをつクラスのメソッドとしてコンパイルされます。のでは、するメインクラスは`my.program.MainKt`です。

のファイルのトップレベルをむクラスのをするには、ファイルのにpackageステートメントのにのアノテーションをします。

```
@file:JvmName("MyApp")
```

このでは、するメインクラスは`my.program.MyApp`なり`my.program.MyApp`。

- `@JvmName`アノテーションをむパッケージレベルの 。
- アノテーションサイトのターゲット

オブジェクトをしたHello World

わりに、KotlinプログラムのmainをむObject Declarationをすることもできます。

```
package my.program

object App {
    @JvmStatic fun main(args: Array<String>) {
        println("Hello World")
    }
```

```
}
```

するクラスはオブジェクトので、このは`my.program.App`です。

このメソッドのトップレベルのは、するクラスがであり、するのがクラス`App`スコープされていることです。また、ステートをしてのをうためのSingletonの`App`インスタンスもあります。

- `@JvmStatic`アノテーションをむメソッド

コンパニオンオブジェクトをしたHello World

オブジェクトとに、クラスのコンパニオンオブジェクトをしてKotlinプログラムの`main`をすることができます。

```
package my.program

class App {
    companion object {
        @JvmStatic fun main(args: Array<String>) {
            println("Hello World")
        }
    }
}
```

するクラスはクラスので、このは`my.program.App`です。

このメソッドのトップレベルのは、するクラスがであり、するのがクラス`App`スコープされていることです。これは、 `Object Declaration`とていますが、それのをうためにクラスをインスタンスすることをしていることはじです。

の "hello"をうためにクラスをインスタンスするわずかなバリエーション

```
class App {
    companion object {
        @JvmStatic fun main(args: Array<String>) {
            App().run()
        }
    }

    fun run() {
        println("Hello World")
    }
}
```

- @JvmStaticアノテーションをむメソッド

varargsをするなメソッド

これらのなメソッドスタイルは、すべてvarargsですることもできます

```
package my.program

fun main(vararg args: String) {
    println("Hello, world!")
}
```

コマンドラインでKotlinコードをコンパイルしてする

javaはJavaコードをコンパイルしするための2つのなるコマンドをしています。 Kotlinとじことはあなたに々のコマンドをします。

javaファイルをコンパイルする`javac`。 `java`はjavaファイルをします。

じ`kotlinc` kotlinファイルをコンパイルするには、 `kotlin`ファイルをkotlinします。

コマンドラインからののみり

コンソールからされたは、Kotlinプログラムでけることができ、としてできます。コマンドプロンプトからN1 2 3などのをすことができます。

Kotlinのコマンドラインのな。

```
fun main(args: Array<String>) {

    println("Enter Two number")
    var (a, b) = readLine()!!.split(' ') // !! this operator use for
NPE(NullPointerException).

    println("Max number is : ${maxNum(a.toInt(), b.toInt())}")
}



fun maxNum(a: Int, b: Int): Int {

    var max = if (a > b) {
        println("The value of a is $a");
        a
    } else {
        println("The value of b is $b")
        b
    }

    return max;

}
```

ここでは、をつけるためにコマンドラインから2つのをします。

```
Enter Two number
71 89 // Enter two number from command line

The value of b is 89
Max number is: 89
```

にとって オペレータNull Safetyをチェックしてください。

のはIntellijでコンパイルしてします。

オンラインでKotlinをいめるをむ https://riptutorial.com/ja/kotlin/topic/490/kotlinをいめる

2: DSLビルディング

き

KotlinのDSLをするためののにしてください。

Examples

DSLをするためのインフィクスアプローチ

あなたがっている

```
infix fun <T> T?.shouldBe(expected: T?) = assertEquals(expected, this)
```

あなたのテストでは、のようなDSLのようなコードをくことができます

```
@Test
fun test() {
  100.plusOne() shouldBe 101
}
```

DSLをするためのinvokeメソッドのオーバーライド

あなたがっている

```
class MyExample(val i: Int) {
  operator fun <R> invoke(block: MyExample.() -> R) = block()
  fun Int.bigger() = this > i
}
```

プロダクションコードにのDSLのようなコードをくことができます

```
fun main2(args: Array<String>) {
    val ex = MyExample(233)
    ex {
        // bigger is defined in the context of `ex`
        // you can only call this method inside this context
        if (777.bigger()) kotlin.io.println("why")
    }
}
```

ラムダでのの

あなたがっている

```
val r = Random(233)
```

```
infix inline operator fun Int.rem(block: () -> Unit) {
  if (r.nextInt(100) < this) block()
}
```

のDSLのようなコードをくことができます

```
20 % { println("The possibility you see this message is 20%") }
```

ラムダでのの

あなたがっている

```
operator fun <R> String.invoke(block: () -> R) = {
  try { block.invoke() }
  catch (e: AssertException) { System.err.println("$this\n${e.message}") }
}
```

のDSLのようなコードをくことができます

```
"it should return 2" {
   parse("1 + 1").buildAST().evaluate() shouldBe 2
}
```

あなたがの`shouldBe`としているとじたら、 `Infix approach to build DSL shouldBe Infix approach to build DSL`のをしてください。

オンラインでDSLビルディングをむ https://riptutorial.com/ja/kotlin/topic/10042/dslビルディング

3: Java 8ストリームにするもの

き

Kotlinは、をするためのコレクションおよびiterableにするくのメソッドをしています。の`Sequence`では、このようなのいくつかをけにすることができます。

について

チェーンをしたいは、チェーンのに`asSequence()`をして`Sequence`できます。ののわりには、、`Sequence`もします。に、 `toList()` 、 `toSet()` 、 `toMap()`またはのをして、に`Sequence`をすることができます。

```
// switch to and from lazy
val someList = items.asSequence().filter { ... }.take(10).map { ... }.toList()

// switch to lazy, but sorted() brings us out again at the end
val someList = items.asSequence().filter { ... }.take(10).map { ... }.sorted()
```

なぜがないのですか

Kotlinのではがされていないことにづくでしょう。これは、Kotlinがなをち、コンパイルになのであるためです。 Javaにはヌルながあり、ろしいNPEをぐのにつので、Javaほどです。それでKotlinでこれは

```
val someList = people.filter { it.age <= 30 }.map { it.name }
```

のものとじです

```
val someList: List<String> = people.filter { it.age <= 30 }.map { it.name }
```

Kotlinはをっているので`people`あり、その`people.age`ある`Int`したがって、フィルタはのみとのができ`Int` 、その`people.name`ある`String`ため、 `map`ステップがする`List<String>` みり`List`の`String`。

、 `List<People>?`ように`people`がおそらく`null`は`List<People>?`に

```
val someList = people?.filter { it.age <= 30 }?.map { it.name }
```

`List<String>?`します`List<String>?` *null*をうためにこの*Kotlin*のなをしてください。また、 *Kotlin*で*nullable*またはのリストを するなもしてください

ストリームの

コトリンでは、それが1されるかどうかは、のタイプにする。 `Sequence`はしいイテレーータをし、「1のみ」をアサートしないり、するたびににリセットすることができます。したがって、Java 8ストリームではのようにしますが、Kotlinではします。

```
// Java:
Stream<String> stream =
Stream.of("d2", "a2", "b1", "b3", "c").filter(s -> s.startsWith("b"));

stream.anyMatch(s -> true);    // ok
stream.noneMatch(s -> true);   // exception
```

```
// Kotlin:
val stream = listOf("d2", "a2", "b1", "b3", "c").asSequence().filter { it.startsWith('b' ) }

stream.forEach(::println) // b1, b2

println("Any B ${stream.any { it.startsWith('b') }}") // Any B true
println("Any C ${stream.any { it.startsWith('c') }}") // Any C false

stream.forEach(::println) // b1, b2
```

そして、Javaではじをる

```
// Java:
Supplier<Stream<String>> streamSupplier =
    () -> Stream.of("d2", "a2", "b1", "b3", "c")
          .filter(s -> s.startsWith("a"));

streamSupplier.get().anyMatch(s -> true);   // ok
streamSupplier.get().noneMatch(s -> true);  // ok
```

したがって、Kotlinでは、データのプロバイダがリセットしてしいイテレーータをするかどうかをします。しかし、に`Sequence`を1のにしたいは、のように`Sequence constrainOnce()`をできます。

```
val stream = listOf("d2", "a2", "b1", "b3", "c").asSequence().filter { it.startsWith('b' ) }
        .constrainOnce()

stream.forEach(::println) // b1, b2
stream.forEach(::println) // Error:java.lang.IllegalStateException: This sequence can be
consumed only once.
```

- Iterableのの APIリファレンス
- のの APIリファレンス
- Listのの APIリファレンス
- Mapへのの APIリファレンス

Examples

リストのをする

```
// Java:
List<String> list = people.stream().map(Person::getName).collect(Collectors.toList());
```

```
// Kotlin:
val list = people.map { it.name }  // toList() not needed
```

をにし、コンマでってする

```
// Java:
String joined = things.stream()
                      .map(Object::toString)
                      .collect(Collectors.joining(", "));
```

```
// Kotlin:
val joined = things.joinToString() // ", " is used as separator, by default
```

の

```
// Java:
int total = employees.stream()
                     .collect(Collectors.summingInt(Employee::getSalary)));
```

```
// Kotlin:
val total = employees.sumBy { it.salary }
```

```
// Java:
Map<Department, List<Employee>> byDept
     = employees.stream()
                .collect(Collectors.groupingBy(Employee::getDepartment));
```

```
// Kotlin:
val byDept = employees.groupBy { it.department }
```

の

```
// Java:
Map<Department, Integer> totalByDept
     = employees.stream()
                .collect(Collectors.groupingBy(Employee::getDepartment,
                     Collectors.summingInt(Employee::getSalary)));
```

```
// Kotlin:
val totalByDept = employees.groupBy { it.dept }.mapValues { it.value.sumBy { it.salary }}
```

をとにする

```
// Java:
Map<Boolean, List<Student>> passingFailing =
```

```
     students.stream()
             .collect(Collectors.partitioningBy(s -> s.getGrade() >= PASS_THRESHOLD));
```

```
// Kotlin:
val passingFailing = students.partition { it.grade >= PASS_THRESHOLD }
```

メンバーの

```
// Java:
List<String> namesOfMaleMembersCollect = roster
    .stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .map(p -> p.getName())
    .collect(Collectors.toList());
```

```
// Kotlin:
val namesOfMaleMembers = roster.filter { it.gender == Person.Sex.MALE }.map { it.name }
```

のメンバーによるグループ

```
// Java:
Map<Person.Sex, List<String>> namesByGender =
      roster.stream().collect(
        Collectors.groupingBy(
            Person::getGender,
            Collectors.mapping(
                Person::getName,
                Collectors.toList())));
```

```
// Kotlin:
val namesByGender = roster.groupBy { it.gender }.mapValues { it.value.map { it.name } }
```

リストをのリストにフィルターする

```
// Java:
List<String> filtered = items.stream()
    .filter( item -> item.startsWith("o") )
    .collect(Collectors.toList());
```

```
// Kotlin:
val filtered = items.filter { item.startsWith('o') }
```

のをする

```
// Java:
String shortest = items.stream()
    .min(Comparator.comparing(item -> item.length()))
    .get();
```

```
// Kotlin:
val shortest = items.minBy { it.length }
```

なるのストリームを2 - のアイテムがけているはする

```
// Java:
Stream.of("a1", "a2", "a3")
    .findFirst()
    .ifPresent(System.out::println);
```

```
// Kotlin:
sequenceOf("a1", "a2", "a3").firstOrNull()?.apply(::println)
```

なるのストリームを3 - のをする

```
// Java:
IntStream.range(1, 4).forEach(System.out::println);
```

```
// Kotlin:  (inclusive range)
(1..3).forEach(::println)
```

なるのストリームを4 - をし、をマップし、をする

```
// Java:
Arrays.stream(new int[] {1, 2, 3})
    .map(n -> 2 * n + 1)
    .average()
    .ifPresent(System.out::println); // 5.0
```

```
// Kotlin:
arrayOf(1,2,3).map { 2 * it + 1}.average().apply(::println)
```

ストリームの5 - のリストをにし、をマップし、Intにし、をつける

```
// Java:
Stream.of("a1", "a2", "a3")
    .map(s -> s.substring(1))
    .mapToInt(Integer::parseInt)
    .max()
    .ifPresent(System.out::println);  // 3
```

```
// Kotlin:
sequenceOf("a1", "a2", "a3")
    .map { it.substring(1) }
    .map(String::toInt)
    .max().apply(::println)
```

ストリームののい6 - Intsのストリームをさせ、をマップし、をする

```
// Java:
IntStream.range(1, 4)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);

// a1
// a2
// a3
```

```
// Kotlin:  (inclusive range)
(1..3).map { "a$it" }.forEach(::println)
```

なるのストリーム**7 -** をにりす、Intにマップする、Stringにマップする、それぞれをする

```
// Java:
Stream.of(1.0, 2.0, 3.0)
    .mapToInt(Double::intValue)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);

// a1
// a2
// a3
```

```
// Kotlin:
sequenceOf(1.0, 2.0, 3.0).map(Double::toInt).map { "a$it" }.forEach(::println)
```

フィルタがされたにリストのアイテムをえる

```
// Java:
long count = items.stream().filter( item -> item.startsWith("t")).count();
```

```
// Kotlin:
val count = items.filter { it.startsWith('t') }.size
// but better to not filter, but count with a predicate
val count = items.count { it.startsWith('t') }
```

ストリームのみ**-** フィルタリング、、リストのべえ

```
// Java:
List<String> myList = Arrays.asList("a1", "a2", "b1", "c2", "c1");

myList.stream()
      .filter(s -> s.startsWith("c"))
      .map(String::toUpperCase)
     .sorted()
     .forEach(System.out::println);

// C1
// C2
```

```
// Kotlin:
val list = listOf("a1", "a2", "b1", "c2", "c1")
list.filter { it.startsWith('c') }.map (String::toUpperCase).sorted()
        .forEach (::println)
```

なるのストリーム1 - のアイテムがあればそれをう

```
// Java:
Arrays.asList("a1", "a2", "a3")
    .stream()
    .findFirst()
    .ifPresent(System.out::println);
```

```
// Kotlin:
listOf("a1", "a2", "a3").firstOrNull()?.apply(::println)
```

または、ifPresentとばれるStringのをします。

```
// Kotlin:
inline fun String?.ifPresent(thenDo: (String)->Unit) = this?.apply { thenDo(this) }

// now use the new extension function:
listOf("a1", "a2", "a3").firstOrNull().ifPresent(::println)
```

`apply()`

?.セーフコールオペレータであり、にnull http : //stackoverflow.com/questions/34498562/in-kotlin-what-is-the-idiomatic-way-to-deal-with-nullable-values-referencing-o/34498563 34498563

コレクション5 - のをしし、フォーマットされたをする

```
// Java:
String phrase = persons
        .stream()
        .filter(p -> p.age >= 18)
        .map(p -> p.name)
        .collect(Collectors.joining(" and ", "In Germany ", " are of legal age."));

System.out.println(phrase);
// In Germany Max and Peter and Pamela are of legal age.
```

```
// Kotlin:
val phrase = persons
        .filter { it.age >= 18 }
        .map { it.name }
        .joinToString(" and ", "In Germany ", " are of legal age.")

println(phrase)
// In Germany Max and Peter and Pamela are of legal age.
```

さらに、Kotlinでは、なデータクラスをし、のようにテストデータをインスタンスできます。

```
// Kotlin:
// data class has equals, hashcode, toString, and copy methods automagically
data class Person(val name: String, val age: Int)

val persons = listOf(Person("Tod", 5), Person("Max", 33),
                     Person("Frank", 13), Person("Peter", 80),
                     Person("Pamela", 18))
```

コレクションの6をする - 、、をにグループする

```
// Java:
Map<Integer, String> map = persons
        .stream()
        .collect(Collectors.toMap(
                p -> p.age,
                p -> p.name,
                (name1, name2) -> name1 + ";" + name2));

System.out.println(map);
// {18=Max, 23=Peter;Pamela, 12=David}
```

それでは、Kotlinにとってもっといケースです。まず、コレクション/シーケンスからMapをするのいをべるったえ

```
// Kotlin:
val map1 = persons.map { it.age to it.name }.toMap()
println(map1)
// output: {18=Max, 23=Pamela, 12=David}
// Result: duplicates overridden, no exception similar to Java 8

val map2 = persons.toMap({ it.age }, { it.name })
println(map2)
// output: {18=Max, 23=Pamela, 12=David}
// Result: same as above, more verbose, duplicates overridden

val map3 = persons.toMapBy { it.age }
println(map3)
// output: {18=Person(name=Max, age=18), 23=Person(name=Pamela, age=23), 12=Person(name=David,
age=12)}
// Result: duplicates overridden again

val map4 = persons.groupBy { it.age }
println(map4)
// output: {18=[Person(name=Max, age=18)], 23=[Person(name=Peter, age=23), Person(name=Pamela,
age=23)], 12=[Person(name=David, age=12)]}
// Result: closer, but now have a Map<Int, List<Person>> instead of Map<Int, String>

val map5 = persons.groupBy { it.age }.mapValues { it.value.map { it.name } }
println(map5)
// output: {18=[Max], 23=[Peter, Pamela], 12=[David]}
// Result: closer, but now have a Map<Int, List<String>> instead of Map<Int, String>
```

それではのために

```
// Kotlin:
val map6 = persons.groupBy { it.age }.mapValues { it.value.joinToString(";") { it.name } }
```

```
println(map6)
// output: {18=Max, 23=Peter;Pamela, 12=David}
// Result: YAY!!
```

はちょうどリストをりたたむとするをするために、するをするためにな`joinToString`からする`Person`にインスタンス`Person.name`。

サンプルの**7a -** マップ、デリミタとの

```
// Java (verbose):
Collector<Person, StringJoiner, String> personNameCollector =
Collector.of(
        () -> new StringJoiner(" | "),          // supplier
        (j, p) -> j.add(p.name.toUpperCase()),  // accumulator
        (j1, j2) -> j1.merge(j2),               // combiner
        StringJoiner::toString);                // finisher

String names = persons
        .stream()
        .collect(personNameCollector);

System.out.println(names);  // MAX | PETER | PAMELA | DAVID

// Java (concise)
String names = persons.stream().map(p -> p.name.toUpperCase()).collect(Collectors.joining(" |
"));
```

```
// Kotlin:
val names = persons.map { it.name.toUpperCase() }.joinToString(" | ")
```

サンプル**7b**をめる **- SummarizingInt**でする

```
// Java:
IntSummaryStatistics ageSummary =
    persons.stream()
           .collect(Collectors.summarizingInt(p -> p.age));

System.out.println(ageSummary);
// IntSummaryStatistics{count=4, sum=76, min=12, average=19.000000, max=23}
```

```
// Kotlin:

// something to hold the stats...
data class SummaryStatisticsInt(var count: Int = 0,
                                var sum: Int = 0,
                                var min: Int = Int.MAX_VALUE,
                                var max: Int = Int.MIN_VALUE,
                                var avg: Double = 0.0) {
    fun accumulate(newInt: Int): SummaryStatisticsInt {
        count++
        sum += newInt
        min = min.coerceAtMost(newInt)
        max = max.coerceAtLeast(newInt)
        avg = sum.toDouble() / count
```

```
        return this
    }
}

// Now manually doing a fold, since Stream.collect is really just a fold
val stats = persons.fold(SummaryStatisticsInt()) { stats, person ->
stats.accumulate(person.age) }

println(stats)
// output: SummaryStatisticsInt(count=4, sum=76, min=12, max=23, avg=19.0)
```

しかし、にはKotlin stdlibのスタイルにマッチする2つのをするがいです

```
// Kotlin:
inline fun Collection<Int>.summarizingInt(): SummaryStatisticsInt
        = this.fold(SummaryStatisticsInt()) { stats, num -> stats.accumulate(num) }

inline fun <T: Any> Collection<T>.summarizingInt(transform: (T)->Int): SummaryStatisticsInt =
        this.fold(SummaryStatisticsInt()) { stats, item -> stats.accumulate(transform(item)) }
```

しい`summarizingInt`をするには、の2つのがあります。

```
val stats2 = persons.map { it.age }.summarizingInt()

// or

val stats3 = persons.summarizingInt { it.age }
```

これらのすべてがじをみします。また、このをして`Sequence`となプリミティブにしてすることもできます。

オンラインでJava 8ストリームにするものをむ https://riptutorial.com/ja/kotlin/topic/707/java-8ストリームにするもの

4: JavaのためのKotlin

き

Kotlinにるほとんどのは、Javaでプログラミングのをっています。

このトピックでは、JavaとKotlinをして、もなとKotlinがJavaでするをするをします。

Examples

の

Kotlinでは、のはJavaのものとしってえます

```
val i : Int = 42
```

- らはどちらかでまる`val`や`var`り、 `final` 「 ヴァル UE 」または**VAR**の iableを。
- タイプはのにされ、 :にられます：
- Kotlinののおかげで、なは、コンパイラがにできるをつ

Java	コトリン
`int i = 42;`	`var i = 42` または `var i : Int = 42`
`final int i = 42;`	`val i = 42`

- コトリンはありません;ステートメントをする
- Kotlinは**null**セーフです
- Kotlinは**100Java**のがあります
- Kotlinにはプリミティブはありませんが であれば、JVMのオブジェクトをします
- Kotlinクラスにはプロパティがあり、フィールドはありません
- Kotlinは、された`equals` / `hashCode`メソッドとフィールドアクセサーをつデータクラスをしています
- Kotlinにはランタイムのみがあり、 チェックはありません
- Kotlinには**`new`**キーワードはありません 。オブジェクトをするには、のメソッドとじようにコンストラクタをびすだけです。
- Kotlinはられた のオーバーロードをサポートしています。たとえば、マップのにアクセスするには、 `val a = someMap["key"]`
- Kotlinは、JVMのバイトコードだけでなく、 **Java Script**にもコンパイルできるだけでなく、Kotlinにバックエンドコードとフロントエンドコードのをくことができます
- Kotlinは、Java 6とにがあります 。これは、それほどいAndroidデバイスのサポートにして

にいものです
- KotlinはAndroidに にサポートされているです
- Kotlinのコレクションには、 なコレクションとなコレクションのにみみのがあります 。
- コトリンはコルーチンをサポートしています

とアイデンティティ

Kotlinはしています`==`のためにつまり、びします`equals`とを`===`アイデンティティのために。

Java	コトリン
`a.equals(b);`	`a == b`
`a == b;`	`a === b`
`a != b;`	`a !== b`

https : //kotlinlang.org/docs/reference/equality.html

IF、TRYなどはであり、ではありません。

Kotlinでは、 `if` 、 `try` 、およびothersはではなくをすため、voidステートメントではありません。

たとえば、KotlinにはJavaの*Elvis*はありませんが、のようにくことができます

```
val i = if (someBoolean) 33 else 42
```

さらによくられていないが、にかなのは、 `try` です。

```
val i = try {
    Integer.parseInt(someString)
}
catch (ex : Exception)
{
    42
}
```

オンラインでJavaのためのKotlinをむ https://riptutorial.com/ja/kotlin/topic/10099/javaのための-kotlin

5: JUnit

Examples

ルール

JUnit ルールをテストフィクスチャにするには

```
@Rule @JvmField val myRule = TemporaryFolder()
```

`@JvmField`アノテーションは、 `myRule`プロパティとじpublicをつバッキングフィールドをするためにです 。 JUnitルールでは、きルールフィールドをするがあります。

オンラインでJUnitをむ https://riptutorial.com/ja/kotlin/topic/6973/junit

6: Kotlin Android

き

Kotlinには、バインディングやButterKnifeなどのフレームワークのをスキップできるように、Androidのビューインジェクションがみまれています。いくつかのは、よりい、よりいなけ、したがってエラーがこりにくいことです。

Examples

にされたgradleプロジェクトからめます 。

プロジェクトのローカル トップレベルではない `build.gradle` 、Kotlinプラグインのにトップレベルのインデントレベルでエクステンションプラグインをします。

```
buildscript {
    ...
}

apply plugin: "com.android.application"
...
apply plugin: "kotlin-android"
apply plugin: "kotlin-android-extensions"
...
```

ビューの

々はとばれるレイアウトでしているとすると、 `activity_main.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <Button
        android:id="@+id/my_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="My button"/>
</LinearLayout>
```

Kotlinのをして、のようなバインディングなしでボタンをびすことができます

```
import kotlinx.android.synthetic.main.activity_main.my_button

class MainActivity: Activity() {
    override fun onCreate(savedInstanceBundle: Bundle?) {
        super.onCreate(savedInstanceBundle)
        setContentView(R.layout.activity_main)
```

```
        // my_button is already casted to a proper type of "Button"
        // instead of being a "View"
        my_button.setText("Kotlin rocks!")
    }
}
```

レイアウトにされるすべてのIDを*でインポートすることもできます

```
// my_button can be used the same way as before
import kotlinx.android.synthetic.main.activity_main.*
```

ビューは、そのレイアウトのアクティビティ/フラグメント/ビューのではできません。

```
import kotlinx.android.synthetic.main.activity_main.my_button

class NotAView {
    init {
        // This sample won't compile!
        my_button.setText("Kotlin rocks!")
    }
}
```

の

AndroidのはのAndroid Product Flavorsでもします。たとえば、 `build.gradle`ようながあるとします。

```
android {
    productFlavors {
        paid {
            ...
        }
        free {
            ...
        }
    }
}
```

そして、えば、フリーフレーバーだけがボタンをっています

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <Button
        android:id="@+id/buy_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Buy full version"/>
</LinearLayout>
```

たちはにフレーバーにバインドすることができます

```
import kotlinx.android.synthetic.free.main_activity.buy_button
```

ビュー―がにされたときにをけるためのをじるリスナー―は、**Kotlin**のでとてもシンプルでらしい

```
mView.afterMeasured {
  // inside this block the view is completely drawn
  // you can get view's height/width, it.height / it.width
}
```

フードの

```
inline fun View.afterMeasured(crossinline f: View.() -> Unit) {
viewTreeObserver.addOnGlobalLayoutListener(object : ViewTreeObserver.OnGlobalLayoutListener {
    override fun onGlobalLayout() {
        if (measuredHeight > 0 && measuredWidth > 0) {
            viewTreeObserver.removeOnGlobalLayoutListener(this)
            f()
        }
    }
})
}
```

オンラインでKotlin Androidをむ https://riptutorial.com/ja/kotlin/topic/9474/kotlin-android

7: KotlinのRecyclerView

き

はちょうどKotlinをしてのしのとRecyclerViewのコードをしたいといます。

Examples

メインクラスとアダプター

はあなたが**Kotlin**のとをっているとしています。に**activity_main.xml**ファイルに**RecyclerView**をし、アダプタクラスをしてください。

```
class MainActivity : AppCompatActivity(){

        lateinit var mRecyclerView : RecyclerView
        val mAdapter : RecyclerAdapter = RecyclerAdapter()

        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            setContentView(R.layout.activity_main)
            val toolbar = findViewById(R.id.toolbar) as Toolbar
            setSupportActionBar(toolbar)

            mRecyclerView = findViewById(R.id.recycler_view) as RecyclerView

            mRecyclerView.setHasFixedSize(true)
            mRecyclerView.layoutManager = LinearLayoutManager(this)
            mAdapter.RecyclerAdapter(getList(), this)
            mRecyclerView.adapter = mAdapter
        }

        private fun getList(): ArrayList<String> {
            var list : ArrayList<String> = ArrayList()
            for  (i in 1..10) { // equivalent of 1 <= i && i <= 10
                println(i)
                list.add("$i")
            }
            return list
        }
    }
```

これはあなたのリサイクラ·ビュー·アダプタ·クラスであり、 **main_item.xml**ファイルをします。

```
class RecyclerAdapter : RecyclerView.Adapter<RecyclerAdapter.ViewHolder>() {

    var mItems: ArrayList<String>  = ArrayList()
    lateinit var mClick : OnClick

    fun RecyclerAdapter(item : ArrayList<String>, mClick : OnClick){
        this.mItems = item
        this.mClick = mClick;
```

```
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val item = mItems[position]
        holder.bind(item, mClick, position)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        val layoutInflater = LayoutInflater.from(parent.context)
        return ViewHolder(layoutInflater.inflate(R.layout.main_item, parent, false))
    }

    override fun getItemCount(): Int {
        return mItems.size
    }

    class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {
        val card = view.findViewById(R.id.card) as TextView
        fun bind(str: String, mClick: OnClick, position: Int){
            card.text = str
            card.setOnClickListener { view ->
                mClick.onClickListner(position)
            }
        }
    }
}
```

オンラインでKotlinのRecyclerViewをむ https://riptutorial.com/ja/kotlin/topic/10143/kotlinのrecyclerview

8: Kotlinのループ

Kotlinでは、ループはなりされたループにコンパイルされます。たとえば、をする、オブジェクトのオーバーヘッドをけるために、バイトコードはなintについてするループにコンパイルされます。

Examples

アクションをxりします。

```
repeat(10) { i ->
    println("This line will be printed 10 times")
    println("We are on the ${i + 1}. loop iteration")
}
```

iterablesをループする

のfor-loopをして、のをループすることができます。

```
val list = listOf("Hello", "World", "!")
for(str in list) {
    print(str)
}
```

Kotlinのくのものは、のようにりしです

```
for(i in 0..9) {
    print(i)
}
```

イテレーションにインデックスがな

```
for((index, element) in iterable.withIndex()) {
    print("$element at index $index")
}
```

forEachをして、なをたずに、ライブラリにまれるへのなアプローチもあります。

```
iterable.forEach {
    print(it.toString())
}
```

`it` 、このでは、に、のをしているラムダを

Whileループ

whileループとdo-whileループはのとじようにします。

```
while(condition) {
    doSomething()
}

do {
    doSomething()
} while (condition)
```

do-whileループでは、ブロックはループでされたとにアクセスできます。

してする

のとじように、キーワードをしてできます。

```
while(true) {
    if(condition1) {
        continue // Will immediately start the next iteration, without executing the rest of
the loop body
    }
    if(condition2) {
        break // Will exit the loop completely
    }
}
```

ネストされたループがあるは、ループステートメントにラベルをけ、breakステートメントとcontinueステートメントをして、どのループをまたはしたいかをできます。

```
outer@ for(i in 0..10) {
    inner@ for(j in 0..10) {
        break       // Will break the inner loop
        break@inner // Will break the inner loop
        break@outer // Will break the outer loop
    }
}
```

しかし、このアプローチはな`forEach`コンストラクトではしません。

kotlinでをする

```
//iterates over a map, getting the key and value at once

var map = hashMapOf(1 to "foo", 2 to "bar", 3 to "baz")

for ((key, value) in map) {
    println("Map[$key] = $value")
}
```

ほとんどのプログラミングのように、Kotlinではをしたループもです。

```
fun factorial(n: Long): Long = if (n == 0) 1 else n * factorial(n - 1)
```

```
println(factorial(10)) // 3628800
```

のでは、 `factorial`は、されたがたされるまでりしびされます。

のための

Kotlinライブラリには、コレクションをりしするためののなもあります。

たとえば、 `map`をしてアイテムのリストをすることができます。

```
val numbers = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 0)
val numberStrings = numbers.map { "Number $it" }
```

このスタイルのくのの1つは、のでをさせることです。もしのためにのリストをフィルタリングするがあったとすれば、ほんのしのしかではありません。 `filter`をすることができます。

```
val numbers = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 0)
val numberStrings = numbers.filter { it % 2 == 0 }.map { "Number $it" }
```

オンラインでKotlinのループをむ https://riptutorial.com/ja/kotlin/topic/2727/kotlinのループ

9: Kotlinビルドの

Examples

グラデル

`kotlin-gradle-plugin`は、GradleでKotlinコードをコンパイルするためにされます。に、そのバージョンはするKotlinバージョンにするがあります。たとえば、Kotlin `1.0.3`をするは、 `kotlin-gradle-plugin` gradle `kotlin-gradle-plugin`バージョン`1.0.3`もです。

`gradle.properties`または`gradle.properties`このバージョンをすることをおめし `ExtraPropertiesExtension`。

```
buildscript {
    ext.kotlin_version = '1.0.3'

    repositories {
      mavenCentral()
    }

    dependencies {
      classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}
```

に、このプラグインをプロジェクトにするがあります。これをうは、なるプラットフォームをターゲットにするときになります。

ターゲティングJVM

```
apply plugin: 'kotlin'
```

Androidをターゲットする

```
apply plugin: 'kotlin-android'
```

JSのターゲット

```
apply plugin: 'kotlin2js'
```

これらはデフォルトのパスです。

- コトリンソース `src/main/kotlin`
- javaソース `src/main/java`

- コトリンテスト `src/test/kotlin`
- java tests `src/test/java`
- ランタイムリソース `src/main/resources`
- テストリソース `src/test/resources`

カスタムプロジェクトレイアウトをしているは、 `SourceSets`をするがあります。

に、プロジェクトにKotlinライブラリのをするがあります

```
dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
}
```

Kotlin Reflectionをするは`compile "org.jetbrains.kotlin:kotlin-reflect:$kotlin_version"`も`compile "org.jetbrains.kotlin:kotlin-reflect:$kotlin_version"`があります。

Android Studioの

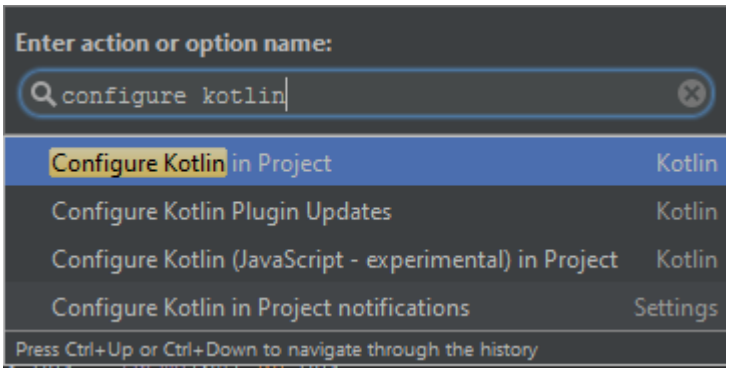
Android Studioでは、Androidプロジェクトでkotlinをにできます。

プラグインをインストールする

Kotlinプラグインをインストールするには、File> Settings> Editor> Plugins> Install JetBrains Plugin ...> Kotlin> Installのにみ、メッセージがされたらAndroid Studioをします。

プロジェクトをする

Android Studioプロジェクトをし、 `Ctrl` + `Shift` + `A`をします。ボックスに「Kotlin in Project」としてEnterをします。

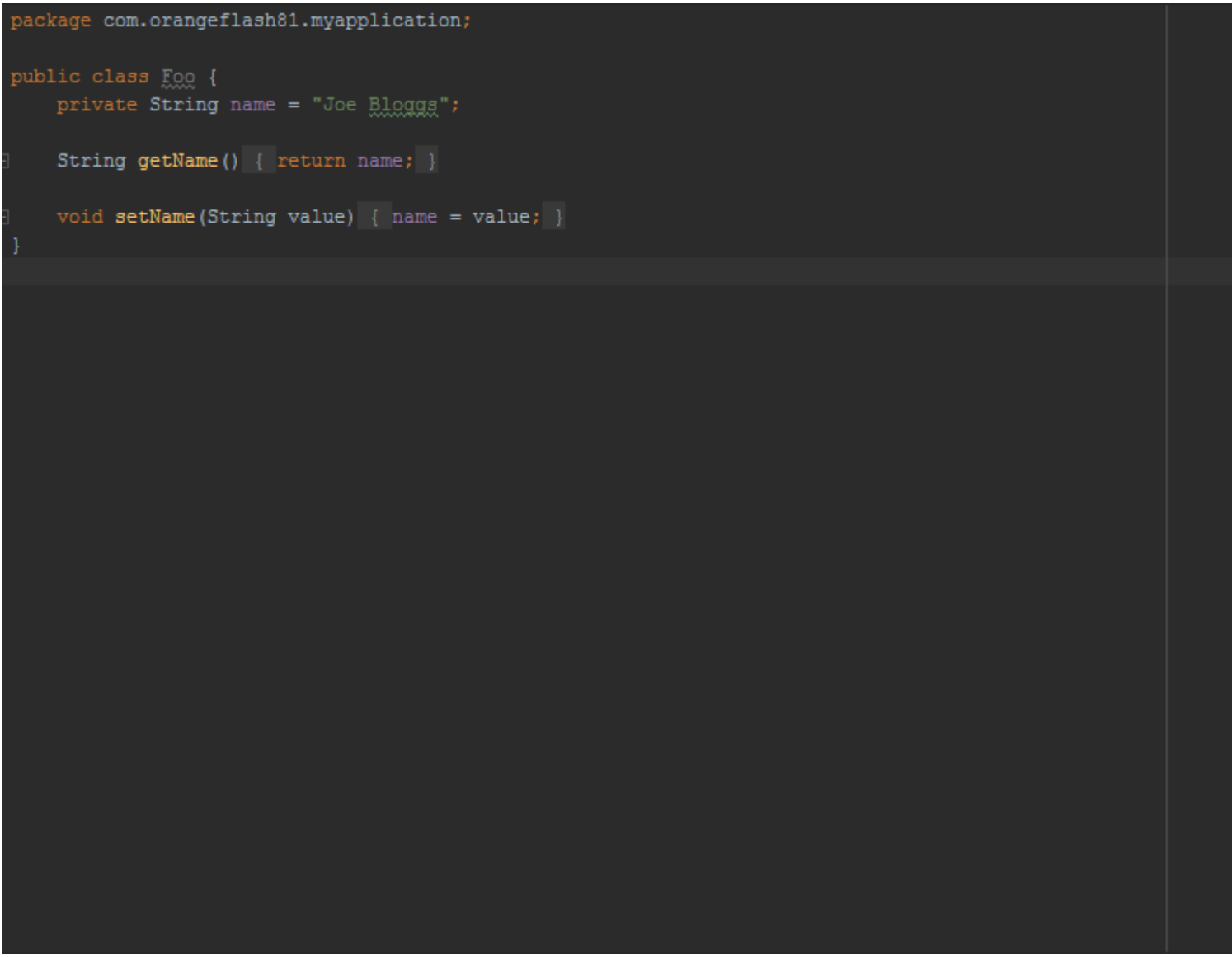


Android Studioはなすべてのをするためにあなたのgradleファイルをします。

Javaの

JavaファイルをKotlinファイルにするには、 `Ctrl` + `Shift` + `A`をして、「JavaファイルをKotlinファ

ィルに] をします。これにより、のファイルのが.ktにされ、コードがKotlinにされます。



GroovyスクリプトをしてGradleからKotlinスクリプトにする

ステップ

- gradle-script-kotlinプロジェクトをする
- されたプロジェクトからプロジェクトにコピー/りけ
    - build.gradle.kts
    - gradlew
    - gradlew.bat
    - settings.gradle
- にじてbuild.gradle.ktsのをすると、クローンされたばかりのプロジェクトのスクリプトやそのサンプルの1つにインスピレーションをえることができます
- Intellijをいてプロジェクトをき、エクスプローラウィンドウでGradleプロジェクトとしてされます。そうでないはにします。

- いた、Intellijがするようにして、 `build.gradle.kts`をき、エラーがないかどうかしてください。ハイライトがしていない、またはのマークがすべていているは、Intellijをじてきます

- Gradleウィンドウをき、それをリフレッシュしてください

Windowsのは、このバグがするがあります。なGradle 3.3ディストリビューションをダウンロードし、されているものをわりにしてください。 。

OSXとUbuntuはすぐにえます。

あなたはすべてのMavenにpublicingのとした、いけたいはのボーナスは、 Jitpackを 、するは、Groovyのにべ、ほぼじです。こののプロジェクトからインスピレーションをることができます。

オンラインでKotlinビルドのをむ https://riptutorial.com/ja/kotlin/topic/2501/kotlinビルドの

10: Nullセーフティ

Examples

NullなとNullableでない

`String`ようなのはnullではありません。それらにヌルをできるようにするには、 `?`それらのにある `String?`

```
var string        : String = "Hello World!"
var nullableString: String? = null

string = nullableString    // Compiler error: Can't assign nullable to non-nullable type.
nullableString = string    // This will work however!
```

セーフコールオペレータ

nullなのとプロパティにアクセスするには、なをするがあります。

のもの、 `?.`アクセスしようとしているプロパティまたはをします。オブジェクトがnullのはnullをします。

```
val string: String? = "Hello World!"
print(string.length)   // Compile error: Can't directly access property of nullable type.
print(string?.length)  // Will print the string's length, or "null" if the string is null.
```

イディオムヌルチェックされたじオブジェクトでのメソッドをびす

ヌルチェックされたオブジェクトののメソッドをびすエレガントなは、Kotlinの`apply`ようにしてい`apply`

```
obj?.apply {
    foo()
    bar()
}
```

これは、びします`foo`と`barobj` ある`this`に`apply`にのみブロック `obj`そうでないはブロックをスキップし、nullです。

とプロパティのびしののにすることなく、ヌルなをnullableでないとしてスコープにれるには、`apply`わりに`let`をうことができ`apply`

```
nullable?.let { notnull ->
    notnull.foo()
```

```
    notnull.bar()
}
```

notnullにをける、あるいはりさとをしてすることができ、のラムダパラメータit。

スマ—トキャスト

あるでオブジェクトがnullにならないとコンパイラができるは、なをもうするはありません。

```
var string: String? = "Hello!"
print(string.length)  // Compile error
if(string != null) {
    // The compiler now knows that string can't be null
    print(string.length)  // It works now!
}
```

> コンパイラは、nullチェックとされたのでされるがあるスマ—トキャストのをしません。
>
> がのブロックのスコ—プからアクセスできるたとえば、ロ—カルオブジェクトのメンバ—なので、スマ—トキャストしてできるしいロ—カルをするがあります。

Iterableとからのヌルをする

にはをCollection<T?>からCollections<T>にするがあります。その、 filterNotNullがたちのソリュ—ションです。

```
val a: List<Int?> = listOf(1, 2, 3, null)
val b: List<Int> = a.filterNotNull()
```

Null Coalescing / Elvis Operator

nullなをif-elseのでることがましいことがあります。エルツ、 ?: 、このようなのためにKotlinでできます。

えば

```
val value: String = data?.first() ?: "Nothing here."
```

っての"Nothing here"data?.first()やdataそのものがnullののdata?.first()

じをしてをスロ—して、コ—ドのをすることもできます。

```
val value: String = data?.second()
    ?: throw IllegalArgumentException("Value can't be null!")
```

> アサ—ションをしてNullPointerExceptionをスロ—することができます
>
> data!!.second()!!

アサーション

`!!`はNULLをし、そののnullのバージョンをします。オブジェクトが`null`、`KotlinNullPointerException`がスローされ`null`。

```
val message: String? = null
println(message!!) //KotlinNullPointerException thrown, app crashes
```

エルビスオペレーター:)

Kotlinでは、 `null reference`をできるをでき`null reference`。 nullな`a`があるとします。「 `a`がnullでないはそれをし、そうでないはnullの`x`する」とうことができます

```
var a: String? = "Nullable String Value"
```

さて、 `a`はnullでもかまいません。したがって、 `a`にアクセスするがある、がまれているかどうかにわらず、チェックをするがあります。このチェックは、の`if...else`ステートメントでできます。

```
val b: Int = if (a != null) a.length else -1
```

しかし、ここではオペレータ`Elvis` エルビス `?:` がる。の`if...else`はElvisでのようにすことができます。

```
val b = a?.length ?: -1
```

`?:`ここでは`a?.length` ののがnullでない`a?.length` 、elvisはそれをします。そうでないは、をここでは`-1` にします。のは、がnullのにのみされます。

オンラインでNullセーフティをむ https://riptutorial.com/ja/kotlin/topic/2080/nullセーフティ

11: インターフェイス

インタフェースのためのKotlinリファレンスドキュメント インタフェース

Examples

インターフェース

Kotlinインタフェースには、メソッドの、およびをできないデフォルトのメソッドがまれています。

```
interface MyInterface {
    fun bar()
}
```

このインターフェイスは、のようにクラスによってできるようになりました。

```
class Child : MyInterface {
   override fun bar() {
       print("bar() was called")
   }
}
```

デフォルトのによるインタフェース

Kotlinのインターフェースは、のデフォルトのをつことができます

```
interface MyInterface {
    fun withImplementation() {
      print("withImplementation() was called")
    }
}
```

そのようなインタフェースをするクラスは、することなくそれらのをすることができます

```
class MyClass: MyInterface {
    // No need to reimplement here
}
val instance = MyClass()
instance.withImplementation()
```

プロパティ

デフォルトのは、プロパティゲッタとセッタにしてもします。

```
interface MyInterface2 {
```

```
    val helloWorld
        get() = "Hello World!"
}
```

インターフェイスアクセサではバッキングフィールドをできません

```
interface MyInterface3 {
    // this property won't compile!
    var helloWorld: Int
        get() = field
        set(value) { field = value }
}
```

の

のインターフェイスがじをしている、またはそれらのすべてが1つのでされている、クラスはなびしをでするがあります

```
interface A {
    fun notImplemented()
    fun implementedOnlyInA() { print("only A") }
    fun implementedInBoth() { print("both, A") }
    fun implementedInOne() { print("implemented in A") }
}

interface B {
    fun implementedInBoth() { print("both, B") }
    fun implementedInOne() // only defined
}

class MyClass: A, B {
    override fun notImplemented() { print("Normal implementation") }

    // implementedOnlyInA() can by normally used in instances

    // class needs to define how to use interface functions
    override fun implementedInBoth() {
        super<B>.implementedInBoth()
        super<A>.implementedInBoth()
    }

    // even if there's only one implementation, there multiple definitions
    override fun implementedInOne() {
        super<A>.implementedInOne()
        print("implementedInOne class implementation")
    }
}
```

インターフェイスのプロパティ

インターフェイスでプロパティをできます。インターフェースはをつことができないので、プロパティをとしてするか、またはアクセサのデフォルトをすることによってのみできます。

```
interface MyInterface {
    val property: Int // abstract

    val propertyWithImplementation: String
        get() = "foo"

    fun foo() {
        print(property)
    }
}

class Child : MyInterface {
    override val property: Int = 29
}
```

デフォルトのでのインタフェースをするの

デフォルトのをむじのメソッドをつのインタフェースをする、どのをするべきかはコンパイラにとってあいまいです。がした、するメソッドをオーバーライドしてカスタムをするがあります。このは、デフォルトのにするかどうかをすることができます。

```
interface FirstTrait {
    fun foo() { print("first") }
    fun bar()
}

interface SecondTrait {
    fun foo() { print("second") }
    fun bar() { print("bar") }
}

class ClassWithConflict : FirstTrait, SecondTrait {
    override fun foo() {
        super<FirstTrait>.foo()  // delegate to the default implementation of FirstTrait
        super<SecondTrait>.foo() // delegate to the default implementation of SecondTrait
    }

    // function bar() only has a default implementation in one interface and therefore is ok.
}
```

スーパーキーワード

```
interface MyInterface {
    fun funcOne() {
        //optional body
        print("Function with default implementation")
    }
}
```

インタフェースのメソッドがのデフォルトのをっているは、superキーワードをしてそのメソッドにアクセスできます。

```
super.funcOne()
```

オンラインでインターフェイスをむ https://riptutorial.com/ja/kotlin/topic/900/インターフェイス

12: クラス

き

Kotlinクラスは、そのインタフェースをするのオブジェクトにメソッドとプロパティをすることによって、インタフェースをできます。これにより、ではなくをしてビヘイビアをするがされます。

Examples

のクラスにメソッドをする

```
interface Foo {
    fun example()
}

class Bar {
    fun example() {
        println("Hello, world!")
    }
}

class Baz(b : Bar) : Foo by b

Baz(Bar()).example()
```

このでは、 `Hello, world!`

オンラインでクラスをむ https://riptutorial.com/ja/kotlin/topic/10575/クラス

13: クラス

き

どんなオブジェクトプログラミングでも、あるのクラスがあります。にさせてください

あなたがフルーツのをプログラムしなければならないとしてください `Apples` 、 `Oranges` と `Pears` 。らはすべてサイズ、、がなります。そのため、たちはなるクラスをっています。

しかし、らのいがでもではないといましょう。にはなく、 `Fruit` ほしいとっていますか `getFruit()` はどのようなりがありますか

えはクラス `Fruit` です。たちはしいクラスをり、それからすべてのをします

- オープン{ベースクラス}
- class {クラス}{クラス}{Init Arguments}
- override {の}
- {DC-Object}は{Base Class}です== true

パラメーター

パラメータ	
ベースクラス	のクラス
クラス	ベースクラスからするクラス
Init	クラスのコンストラクタにされる
	クラスのコードとなるコードをつクラスの
DCオブジェクト	クラスのをつ「クラスオブジェクト」オブジェクト

Examples

'open'キーワード

Kotlinでは、クラスはデフォルトでです 。つまり、クラスはできません。

クラスのをするには、 `open` キーワードをし `open` 。

```
open class Thing {
    // I can now be extended!
```

```
}
```

クラス、クラス、およびインタフェースは、デフォルトで`open`れます。

クラスからフィールドをする

クラスの

```
open class BaseClass {
    val x = 10
}
```

クラスの

```
class DerivedClass: BaseClass() {
    fun foo() {
        println("x is equal to " + x)
    }
}
```

サブクラスの

```
fun main(args: Array<String>) {
    val derivedClass = DerivedClass()
    derivedClass.foo() // prints: 'x is equal to 10'
}
```

クラスからメソッドをする

クラスの

```
open class Person {
    fun jump() {
        println("Jumping...")
    }
}
```

クラスの

```
class Ninja: Person() {
    fun sneak() {
        println("Sneaking around...")
    }
}
```

は**Person**のすべてのメソッドにアクセスできます

```
fun main(args: Array<String>) {
    val ninja = Ninja()
    ninja.jump()  // prints: 'Jumping...'
    ninja.sneak() // prints: 'Sneaking around...'
}
```

プロパティとメソッドのオーバーライド

プロパティのオーバーライドみりとの

```
abstract class Car {
    abstract val name: String;
    open var speed: Int = 0;
}

class BrokenCar(override val name: String) : Car() {
    override var speed: Int
        get() = 0
        set(value) {
            throw UnsupportedOperationException("The car is bloken")
        }
}

fun main(args: Array<String>) {
    val car: Car = BrokenCar("Lada")
    car.speed = 10
}
```

メソッドのオーバーライド

```
interface Ship {
    fun sail()
    fun sink()
}

object Titanic : Ship {

    var canSail = true

    override fun sail() {
        sink()
    }

    override fun sink() {
        canSail = false
    }
}
```

オンラインでクラスをむ https://riptutorial.com/ja/kotlin/topic/5622/クラス

14: コトリンにログインする

する Kotlinにログインするな

Examples

kotlin.logging

```
class FooWithLogging {
  companion object: KLogging()

  fun bar() {
    logger.info { "hello $name" }
  }

  fun logException(e: Exception) {
    logger.error(e) { "Error occured" }
  }
}
```

kotlin.loggingフレームワークの

オンラインでコトリンにログインするをむ https://riptutorial.com/ja/kotlin/topic/3258/コトリンにログインする

15: コトリンの

き

このトピックでは、けのKotlinのについてします。

1. Kotlinファイルのは.ktです。
2. Kotlinのすべてのクラスには、のスーパークラスAnyがあります。これは、スーパータイプがされていないクラスJavaのObjectにていますのデフォルトのスーパーです。
3. はvalimmutable- assign onceまたはvarmutables- valueはとしてできます。
4. ステートメントのにセミコロンはありません。
5. がなをさない、そのりのはUnitです。これもオプションです。 6.は===によってチェックされます。 a === bは、aとbがじオブジェクトをしているにのみとされます。

Examples

な

1.ユニットりは、ではオプションです。のコードはです。

```
    fun printHello(name: String?): Unit {
        if (name != null)
            println("Hello ${name}")
    }

    fun printHello(name: String?) {
        ...
    }
```

2.Single-Expressionがのをすとき、はでき、bodyは= symbolのにされます

```
  fun double(x: Int): Int = x * 2
```

これがコンパイラによってできる、りのをにすることはオプションです

```
  fun double(x: Int) = x * 2
```

3.をうのはです。

```
In java:
    int num=10
    String s  = "i =" + i;

In Kotlin
    val num = 10
    val s = "i = $num"
```

4.In Kotlinでは、システムは、nullnullableをできると、nullのできないをします。たとえば、Stringののはnullをできません。

```
var a: String = "abc"
a = null // compilation error
```

nullをするには、をnullなとしてすることができます。

```
var b: String? = "abc"
b = null // ok
```

5.In Kotlinでは、==はにのしいかどうかをチェックします。では、a == bのようなはのようにされます。

```
    a?.equals(b) ?: (b === null)
```

オンラインでコトリンのをむ https://riptutorial.com/ja/kotlin/topic/10648/コトリンの

16: コトリン

Examples

nullなにしてtoStringをびす

Kotlinで`toString`メソッドをするときにすべきは、 `String?`とみわせてnullをすること`String?`。

たとえば、Androidの`EditText`からテキストをしたいとします。

のようなコーードがあります。

```
// Incorrect:
val text = view.textField?.text.toString() ?: ""
```

フィーールドがしない、はのになりますが、このは`"null"`ます。

```
// Correct:
val text = view.textField?.text?.toString() ?: ""
```

オンラインでコトリンをむ https://riptutorial.com/ja/kotlin/topic/6608/コトリン

17: コルーチン

き

コトルのコルーチンのまだの

Examples

ブロックではなく1れたシンプルなコルーチン

から

```
fun main(args: Array<String>) {
    launch(CommonPool) { // create new coroutine in common thread pool
        delay(1000L) // non-blocking delay for 1 second (default time unit is ms)
        println("World!") // print after delay
    }
    println("Hello,") // main function continues while coroutine is delayed
    Thread.sleep(2000L) // block main thread for 2 seconds to keep JVM alive
}
```

```
Hello,
World!
```

オンラインでコルーチンをむ https://riptutorial.com/ja/kotlin/topic/10936/コルーチン

18: コレクション

き

くのとはなり、Kotlinはなコレクションとなコレクションリスト、セット、マップなどをします。コレクションをいつできるかをにすることは、バグをしたり、れたAPIをするのにちます。

- listOf、mapOfおよびsetOfは、アイテムをまたはできないみりオブジェクトをします。
- をまたはするは、arrayListOf、hashMapOf、hashSetOf、linkedMapOfLinkedHashMap、linkedSetOfLinkedHashSet、mutableListOfKotlin MultableListコレクション、mutableMapOf Kotlin MultableMapコレクション、mutableSetOfKotlin MultibleSetコレクション、sortedMapOfまたはsortedSetOf
- コレクションには、first、last、get、filter、map、join、reduceなどのラムダなどのメソッドがあります。

Examples

リストをう

```
// Create a new read-only List<String>
val list = listOf("Item 1", "Item 2", "Item 3")
println(list) // prints "[Item 1, Item 2, Item 3]"
```

マップの

```
// Create a new read-only Map<Integer, String>
val map = mapOf(Pair(1, "Item 1"), Pair(2, "Item 2"), Pair(3, "Item 3"))
println(map) // prints "{1=Item 1, 2=Item 2, 3=Item 3}"
```

セットをう

```
// Create a new read-only Set<String>
val set = setOf(1, 3, 5)
println(set) // prints "[1, 3, 5]"
```

オンラインでコレクションをむ https://riptutorial.com/ja/kotlin/topic/8846/コレクション

19: ジェネリックス

き

リストには、、、またはにかをすることができます。だからたちはリストをとんでいます。

ジェネリックスは、には、クラスができるとオブジェクトがしているをするためにされます。

- クラス*ClassName* < ***TypeName*** >
- クラス*ClassName* <*>
- *ClassName* <in ***UpperBound*** >
- *ClassName* <out ***LowerBound*** >
- クラス < ***TypeName*** ***UpperBound*** >

パラメーター

パラメータ	
TypeName	タイプパラメータの
LowerBound	コントラバナント
クラス	クラスの

するはNullableです

Kotlinジェネリックスでは、パラメータ`T`は`Any?`。したがって、このクラスの

```
class Consumer<T>
```

パラメータ`T`はに`T: Any?`。nullのをるには、にの`T: Any`をいます。えば

```
class Consumer<T: Any>
```

Examples

サイトの

サイトのは、サイトののとみなすことができます。

```
class Consumer<in T> { fun consume(t: T) { ... } }

fun charSequencesConsumer() : Consumer<CharSequence>() = ...

val stringConsumer : Consumer<String> = charSequenceConsumer() // OK since in-projection
val anyConsumer : Consumer<Any> = charSequenceConsumer() // Error, Any cannot be passed

val outConsumer : Consumer<out CharSequence> = ... // Error, T is `in`-parameter
```

サイトののなは、`List<out T>`であり、これは`T`がりのとしてのみれるようにであり、`Comparator<in T>`はとして`T`をけるだけです。

サイト

サイトのは、Javaワイルドカードにています。

アウトプロジェクション

```
val takeList : MutableList<out SomeType> = ... // Java: List<? extends SomeType>

val takenValue : SomeType = takeList[0] // OK, since upper bound is SomeType

takeList.add(takenValue) // Error, lower bound for generic is not specified
```

ィンプロジェクション

```
val putList : MutableList<in SomeType> = ... // Java: List<? super SomeType>

val valueToPut : SomeType = ...
putList.add(valueToPut) // OK, since lower bound is SomeType

putList[0] // This expression has type Any, since no upper bound is specified
```

```
val starList : MutableList<*> = ... // Java: List<?>

starList[0] // This expression has type Any, since no upper bound is specified
starList.add(someValue) // Error, lower bound for generic is not specified
```

- Variant JavaからKotlinをびすときの。

オンラインでジェネリックスをむ https://riptutorial.com/ja/kotlin/topic/1147/ジェネリックス

20: シングルトンオブジェクト

き

オブジェクトはなのクラスで、 `object`キーワードをしてすることができます。オブジェクトはJavaのSingletonデザインパターンにています。これはjavaのなとしてもします。 Javaからkotlinにりえるは、な、またはシングルトンのわりに、このをにできます。

Examples

Javaのメソッド/フィールドのrepalcementとして

```
object CommonUtils {

    var anyname: String ="Hello"

    fun dispMsg(message: String) {
        println(message)
    }
}
```

のクラスから、とをこのようにびすだけです

```
CommonUtils.anyname
CommonUtils.dispMsg("like static call")
```

シングルトンとしてする

にはKotlinオブジェクトはなるシングルトンです。そのなは、シングルトンのインスタンスをするために`SomeSingleton.INSTANCE`をするがないことです。

javaでは、シングルトンはのようになります。

```
public enum SharedRegistry {
    INSTANCE;
    public void register(String key, Object thing) {}
}

public static void main(String[] args) {
    SharedRegistry.INSTANCE.register("a", "apple");
    SharedRegistry.INSTANCE.register("b", "boy");
    SharedRegistry.INSTANCE.register("c", "cat");
    SharedRegistry.INSTANCE.register("d", "dog");
}
```

kotlinでは、コードはのようになります。

```
object SharedRegistry {
    fun register(key: String, thing: Object) {}
}

fun main(Array<String> args) {
    SharedRegistry.register("a", "apple")
    SharedRegistry.register("b", "boy")
    SharedRegistry.register("c", "cat")
    SharedRegistry.register("d", "dog")
}
```

それはobvoiuslyあまりなすることです。

オンラインでシングルトンオブジェクトをむ https://riptutorial.com/ja/kotlin/topic/10152/シングルトンオブジェクト

21: タイプエイリアス

き

エイリアスでは、のにエイリアスをけることができます。これは、 `(String) -> Boolean`や `Pair<Person, Person>`ようなのようなにをけるのにです。

タイプエイリアスはジェネリックをサポートします。エイリアスはタイプをジェネリックにきえることができ、エイリアスはジェネリックにすることができます。

- タイプアリアス エイリアス = タイプ

エイリアスはコンパイラのです。 JVMのコードにはもされません。すべてのエイリアスはにきえられます。

Examples

```
typealias StringValidator = (String) -> Boolean
typealias Reductor<T, U, V> = (T, U) -> V
```

タイプ

```
typealias Parents = Pair<Person, Person>
typealias Accounts = List<Account>
```

オンラインでタイプエイリアスをむ https://riptutorial.com/ja/kotlin/topic/9453/タイプエイリアス

22: タイプセーフビルダ

タイプセーフなビルダーは、ではなくコンセプトなので、にされていません。

タイプセーフなビルダーのな

のビルダーは、3つのステップでされます。

1. オブジェクトをします。
2. ラムダをしてオブジェクトをします。
3. するオブジェクトをするかす。

Kotlinライブラリのされたビルダー

タイプセーフなビルダーのコンセプトは、Kotlinのライブラリやフレームワークでくわれています。

- アンコ
- わさび
- Ktor
-

Examples

なツリービルダ

ビルダーは、をとするラムダをるのセットとしてできます。このでは、 `JFrame`メニューがされています

```
import javax.swing.*

fun JFrame.menuBar(init: JMenuBar.() -> Unit) {
    val menuBar = JMenuBar()
    menuBar.init()
    setJMenuBar(menuBar)
}

fun JMenuBar.menu(caption: String, init: JMenu.() -> Unit) {
    val menu = JMenu(caption)
    menu.init()
    add(menu)
}

fun JMenu.menuItem(caption: String, init: JMenuItem.() -> Unit) {
    val menuItem = JMenuItem(caption)
    menuItem.init()
    add(menuItem)
```

```
}
```

これらのをすると、オブジェクトのツリーをにできます。

```
class MyFrame : JFrame() {
    init {
        menuBar {
            menu("Menu1") {
                menuItem("Item1") {
                    // Initialize MenuItem with some Action
                }
                menuItem("Item2") {}
            }
            menu("Menu2") {
                menuItem("Item3") {}
                menuItem("Item4") {}
            }
        }
    }
}
```

オンラインでタイプセーフビルダをむ https://riptutorial.com/ja/kotlin/topic/6010/タイプセーフビルダ

23: レンジ

き

は、inとinでされるの..をつrangeToでされます。はすべてのなにしてされますが、のプリミティブではされたがあります

Examples

の

インテグラルのIntRange、LongRange、CharRangeにはながあります。それらはがです。コンパイラはこれを、Javaのインデックスきfor-loopとに、なオーバーヘッドなしにします

```
for (i in 1..4) print(i) // prints "1234"
for (i in 4..1) print(i) // prints nothing
```

downTo

をにりしたいはそれはです。ライブラリでされたdownToをすることができます

```
for (i in 4 downTo 1) print(i) // prints "4321"
```

step

のステップでをすることはですか1にしくはありませんかもちろん、stepはあなたにちます

```
for (i in 1..4 step 2) print(i) // prints "13"
for (i in 4 downTo 1 step 2) print(i) // prints "42"
```

まで

endをまないをするには、untilをします。

```
for (i in 1 until 10) { // i in [1, 10), 10 is excluded
println(i)
}
```

オンラインでレンジをむ https://riptutorial.com/ja/kotlin/topic/10121/レンジ

24:

Examples

try-catch-finallyをってをキャッチする

Kotlinのをキャッチするのは Javaとによくています

```
try {
    doSomething()
}
catch(e: MyException) {
    handle(e)
}
finally {
    cleanup()
}
```

のをキャッチすることもできます

```
try {
    doSomething()
}
catch(e: FileSystemException) {
    handle(e)
}
catch(e: NetworkException) {
    handle(e)
}
catch(e: MemoryException) {
    handle(e)
}
finally {
    cleanup()
}
```

`try`もであり、をすがあります

```
val s: String? = try { getString() } catch (e: Exception) { null }
```

Kotlinはをチエックしていないので、をキャッチするはありません。

```
fun fileToString(file: File) : String {
    //readAllBytes throws IOException, but we can omit catching it
    fileContent = Files.readAllBytes(file)
    return String(fileContent)
}
```

オンラインでをむ https://riptutorial.com/ja/kotlin/topic/7246/

25:

Javaのとに、Kotlinのenumクラスは、されたenumをリストし、そのでをることをにするメソッドをっています。これらのメソッドのシグネチャはのとおりですenumクラスのが`EnumClass`と`EnumClass`。

```
EnumClass.valueOf(value: String): EnumClass
EnumClass.values(): Array<EnumClass>
```

されたがクラスでされているのいずれともしない、 `valueOf()`メソッドは`IllegalArgumentException`スローします。

すべてのenumには、enumクラスのとをするためのプロパティがあります。

```
val name: String
val ordinal: Int
```

enumは、Comparableインタフェースもしています。は、enumクラスでされているです。

Examples

のクラスとじようにEnumクラスはコンストラクタをち、されます

```
enum class Color(val rgb: Int) {
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF)
}
```

enumsのとプロパティ

クラスはメンバつまり、プロパティとをすることもできます。のenumオブジェクトとのメンバーのにセミコロン ; をするがあります。

メンバーが`abstract`、enumオブジェクトはそれをするがあります。

```
enum class Color {
    RED {
        override val rgb: Int = 0xFF0000
    },
    GREEN {
        override val rgb: Int = 0x00FF00
    },
    BLUE {
        override val rgb: Int = 0x0000FF
    }

    ;
```

```
    abstract val rgb: Int

    fun colorString() = "#%06X".format(0xFFFFFF and rgb)
}
```

なenum

```
enum class Color {
  RED, GREEN, BLUE
}
```

enumはオブジェクトです。 Enumはコンマでられています。

はですが、これはシングルトンのをするのです。

```
enum class Planet(var population: Int = 0) {
    EARTH(7 * 100000000),
    MARS();

    override fun toString() = "$name[population=$population]"
}

 println(Planet.MARS) // MARS[population=0]
 Planet.MARS.population = 3
 println(Planet.MARS) // MARS[population=3]
```

オンラインでをむ https://riptutorial.com/ja/kotlin/topic/2286/

26:

き

リフレクションとは、コンパイルではなくにコードをするのです。

リフレクションは、にクラスおよびをイントロスペクションするためのメカニズムです。

JVMプラットフォームをとする、リフレクションは々のJAR `kotlin-reflect.jar`ます。これはのサイズをらし、のをカットし、のJSのようなプラットフォームをターゲットにすることをにするためにわれます。

Examples

クラスの

いくつかのクラスをす`KClass`オブジェクトへのをするには、コロンをします。

```
val c1 = String::class
val c2 = MyClass::class
```

の

はKotlinのです。コロンをしてをし、のにすことができます。

```
fun isPositive(x: Int) = x > 0

val numbers = listOf(-2, -1, 0, 1, 2)
println(numbers.filter(::isPositive)) // [1, 2]
```

Javaリフレクションとの

Kotlinの`KClass`からJavaの`Class`オブジェクトをするには、 `.java`プロパティをします。

```
val stringKClass: KClass<String> = String::class
val c1: Class<String> = stringKClass.java

val c2: Class<MyClass> = MyClass::class.java
```

のは、の`KClass`インスタンスをりてないようにコンパイラによってされます。

クラスのすべてのプロパティのをする

えられた`Example`クラス`BaseExample`いくつかのプロパティをつクラスを

```
open class BaseExample(val baseField: String)

class Example(val field1: String, val field2: Int, baseField: String):
    BaseExample(baseField) {

    val field3: String
        get() = "Property without backing field"

    val field4 by lazy { "Delegated value" }

    private val privateField: String = "Private value"
}
```

クラスのすべてのプロパティをすることができます

```
val example = Example(field1 = "abc", field2 = 1, baseField = "someText")

example::class.memberProperties.forEach { member ->
    println("${member.name} -> ${member.get(example)}")
}
```

このコードをすると、がスローされます。プロパティ`private val privateField`はprivateとされ、そので`member.get(example)`をびすとしません。これをしてプライベートプロパティをする1つの。これをうには、プロパティのJavaゲッターのをチェックするがあります。 `private val`の、getterはしないため、プライベートアクセスがです。

ヘルパーとそのはのようになります

```
fun isFieldAccessible(property: KProperty1<*, *>): Boolean {
    return property.javaGetter?.modifiers?.let { !Modifier.isPrivate(it) } ?: false
}

val example = Example(field1 = "abc", field2 = 1, baseField = "someText")

example::class.memberProperties.filter { isFieldAccessible(it) }.forEach { member ->
    println("${member.name} -> ${member.get(example)}")
}
```

のアプローチは、リフレクションをしてプライベートプロパティにアクセスできるようにすることです。

```
example::class.memberProperties.forEach { member ->
    member.isAccessible = true
    println("${member.name} -> ${member.get(example)}")
}
```

クラスのすべてのプロパティのをする

として、サンプルクラスのすべてのプロパティをする

```
class TestClass {
    val readOnlyProperty: String
```

```
        get() = "Read only!"

    var readWriteString = "asd"
    var readWriteInt = 23

    var readWriteBackedStringProperty: String = ""
        get() = field + '5'
        set(value) { field = value + '5' }

    var readWriteBackedIntProperty: Int = 0
        get() = field + 1
        set(value) { field = value - 1 }

    var delegatedProperty: Int by TestDelegate()

    private var privateProperty = "This should be private"

    private class TestDelegate {
        private var backingField = 3

        operator fun getValue(thisRef: Any?, prop: KProperty<*>): Int {
            return backingField
        }

        operator fun setValue(thisRef: Any?, prop: KProperty<*>, value: Int) {
            backingField += value
        }
    }
}
```

なプロパティをするには、すべてのプロパティをし、なプロパティをタイプにフィルタリングします。プライベートプロパティをみるとがするため、もチェックするがあります。

```
val instance = TestClass()
TestClass::class.memberProperties
        .filter{ prop.visibility == KVisibility.PUBLIC }
        .filterIsInstance<KMutableProperty<*>>()
        .forEach { prop ->
            System.out.println("${prop.name} -> ${prop.get(instance)}")
        }
```

すべての`String`プロパティを`"Our Value"`にするには、りのでさらにフィルタリングできます。KotlinはJava VMをベースにしているため、タイプがなので、`List<String>`などのタイプをすプロパティは`List<Any>`とじになります。ながら、はのではなく、これをけるためのなはないので、あなたのでするがあります。

```
val instance = TestClass()
TestClass::class.memberProperties
        .filter{ prop.visibility == KVisibility.PUBLIC }
        // We only want strings
        .filter{ it.returnType.isSubtypeOf(String::class.starProjectedType) }
        .filterIsInstance<KMutableProperty<*>>()
        .forEach { prop ->
            // Instead of printing the property we set it to some value
            prop.setter.call(instance, "Our Value")
        }
```

オンライ ンでをむ https://riptutorial.com/ja/kotlin/topic/2402/

27:

き

Kotlinには、4のがあります。

これはどこからでもアクセスできます。

これはモジュールコードからのみアクセスできます。

Protectedこれは、それをするクラスとのクラスからのみアクセスできます。

これはそれをするクラスのスコープからのみアクセスできます。

- `<visibility modifier> val/var <variable name> = <value>`

Examples

コードサンプル

パブリック `public val name = "Avijit"`

プライベート `private val name = "Avijit"`

された `protected val name = "Avijit"`

```
internal val name = "Avijit"
```

オンラインでをむ https://riptutorial.com/ja/kotlin/topic/10019/

28: なラムダ

- なパラメ―タ
- {parameterNameParameterType、otherParameterNameOtherParameterType - > anExpression}
- されるパラメ―タ
- Int、Int - > Int = {a、b - > a + b}
- のパラメ―タは、 `it` します。
- val squareInt - > Int = {it * it}
-
- - > ResultType
- InputType - > ResultType
- InputType1、InputType2 - > ResultType

のパラメ―タは、コンテキストからできるときにすることができます。たとえば、をとるクラスのがあるとします。

```
data class User(val fistName: String, val lastName: String) {
    fun username(userNameGenerator: (String, String) -> String) =
        userNameGenerator(firstName, secondName)
}
```

ラムダをすことによってこのをすることができます。パラメ―タはすでにシグネチャでされているため、ラムダでパラメ―タをするはありません。

```
val user = User("foo", "bar")
println(user.userName { firstName, secondName ->
     "${firstName.toUppercase}"_"${secondName.toUppercase}"
 }) // prints FOO_BAR
```

これは、ラムダをにするにもされます。

```
//valid:
val addition: (Int, Int) = { a, b -> a + b }
//valid:
val addition = { a: Int, b: Int -> a + b }
//error (type inference failure):
val addition = { a, b -> a + b }
```

ラムダは、つのパラメ―タをり、はからできるは、のでパラメ―タをすることができ `it` 。

```
listOf(1,2,3).map { it * 2 } // [2,4,6]
```

Examples

をフィルタリングするパラメータとしてのLambda

```
val allowedUsers = users.filter { it.age > MINIMUM_AGE }
```

ラムダはとしてされる

```
val isOfAllowedAge = { user: User -> user.age > MINIMUM_AGE }
val allowedUsers = users.filter(isOfAllowedAge)
```

びしをベンチマークするためのLambda

のにするをするストップウォッチ

```
object Benchmark {
    fun realtime(body: () -> Unit): Duration {
        val start = Instant.now()
        try {
            body()
        } finally {
            val end = Instant.now()
            return Duration.between(start, end)
        }
    }
}
```

```
val time = Benchmark.realtime({
    // some long-running code goes here ...
})
println("Executed the code in $time")
```

オンラインでなラムダをむ https://riptutorial.com/ja/kotlin/topic/5878/なラムダ

29: されたプロパティ

き

Kotlinはプロパティのをハンドラオブジェクトにできます。やなプロパティなど、いくつかのハンドラがまれています。カスタムハンドラもできます。

Examples

い

```
val foo : Int by lazy { 1 + 1 }
println(foo)
```

このでは、 2されます。

な

```
var foo : Int by Delegates.observable("1") { property, oldValue, newValue ->
    println("${property.name} was changed from $oldValue to $newValue")
}
foo = 2
```

```
foo was changed from 1 to 2
```

にづくプロパティ

```
val map = mapOf("foo" to 1)
val foo : String by map
println(foo)
```

このでは、 1

カスタム

```
class MyDelegate {
    operator fun getValue(owner: Any?, property: KProperty<*>): String {
        return "Delegated value"
    }
}

val foo : String by MyDelegate()
println(foo)
```

このでは、 Delegated valueしDelegated value

デリゲートをらすためのレイヤーとしてできます。

KotlinのNullシステムと`WeakReference<T>`えてみましょう。

だから、らかのリファレンスをしなければならないとしましょう。メモリリークをけたいのですが、 `WeakReference`がどこにるかはここです。

たとえばこれをる

```
class MyMemoryExpensiveClass {
    companion object {
        var reference: WeakReference<MyMemoryExpensiveClass>? = null

        fun doWithReference(block: (MyMemoryExpensiveClass) -> Unit) {
            reference?.let {
                it.get()?.let(block)
            }
        }
    }

    init {
        reference = WeakReference(this)
    }
}
```

これは1つのWeakReferenceだけです。このボイラープレートをらすために、カスタムプロパティデリゲートをして、のようにすることができます。

```
class WeakReferenceDelegate<T>(initialValue: T? = null) : ReadWriteProperty<Any, T?> {
    var reference = WeakReference(initialValue)
        private set

    override fun getValue(thisRef: Any, property: KProperty<*>): T? = reference.get()

    override fun setValue(thisRef: Any, property: KProperty<*>, value: T?) {
        reference = WeakReference(value)
    }
}
```

これで、 `WeakReference`ラップされたを、のnullableとにできます。

```
class MyMemoryExpensiveClass {
    companion object {
        var reference: MyMemoryExpensiveClass? by
WeakReferenceDelegate<MyMemoryExpensiveClass>()

        fun doWithReference(block: (MyMemoryExpensiveClass) -> Unit) {
            reference?.let(block)
        }
    }

    init {
        reference = this
    }
}
```

オンラインでされたプロパティをむ https://riptutorial.com/ja/kotlin/topic/10571/されたプロパティ

30:

Examples

DTOのPOJO / POCO

kotlinのデータクラスは、データをするだけでするためにされたクラスです。そのようなクラスは`data`としてマークされ`data`

```
data class User(var firstname: String, var lastname: String, var age: Int)
```

のコードは`User`クラスをし、のコードをにします

- すべてのプロパティのゲッターとセッター`val`のゲッターのみ
- `equals()`
- `hashcode()`
- `toString()`
- `copy()`
- `componentN()` `N`はのにするプロパティ

とに、デフォルトもできます。

```
data class User(var firstname: String = "Joe", var lastname: String = "Bloggs", var age: Int =
20)
```

は、ここでデータクラスをしてください。

リストのフィルタリング

```
val list = listOf(1,2,3,4,5,6)

//filter out even numbers

val even = list.filter { it % 2 == 0 }

println(even) //returns [2,4]
```

のコンストラクタでクラスをせずにクラスにする

あるクラスにしたいが、コンストラクターパラメーターにdelegated-toクラスをしたくないとします。わりに、それをにして、コンストラクタのびしにそれをらさないようにしたいとします。は、クラスでコンストラクタパラメータのみをできるため、これはにえるかもしれません。しかし、 このえにあるように、それをうがあります

```
class MyTable private constructor(table: Table<Int, Int, Int>) : Table<Int, Int, Int> by table
{
```

```
    constructor() : this(TreeBasedTable.create()) // or a different type of table if desired
}
```

これで`MyTable`のコンストラクタを`MyTable()`ようにびすことができます。 `MyTable`する`Table<Int, Int, Int>`、にされます。コンストラクター—のびしはもらない。

このはこのSOのについています。

KotlinのSerializableとserialVersionUid

Kotlinでクラスの`serialVersionUID`をするには、クラスのコンパニオンオブジェクトにメンバー—をするといういくつかのオプションがあります。

もなバイトコ—ドは、するクラスこのは`MySpecialCase`プライベ—トになる`private const val`から`MySpecialCase`ます。

```
class MySpecialCase : Serializable {
    companion object {
        private const val serialVersionUID: Long = 123
    }
}
```

これらのフォ—ムをすることもできます。これらのフォ—ムはそれぞれのためにでないゲッター—/セッター—メソッドをつというをいます。

```
class MySpecialCase : Serializable {
    companion object {
        private val serialVersionUID: Long = 123
    }
}
```

これにより、フィ—ルドがされますが、 `getSerialVersionUID`もなコンパニオンオブジェクトにされます。

```
class MySpecialCase : Serializable {
    companion object {
        @JvmStatic private val serialVersionUID: Long = 123
    }
}
```

これにより、フィ—ルドがされますが、ゲッター—と`getSerialVersionUID`がまれているクラス`MySpecialCase`にもですがされます。

しかし、すべては`serialVersionUID`を`Serializable`クラスにするとしてします。

Kotlinのなメソッド

Kotlinのなメソッドは、Javaとじにすることができます

```
fun doSomething() {
   someOtherAction()
   return this
}
```

ただし、のようなをすることで、よりにすることもできます。

```
fun <T: Any> T.fluently(func: ()->Unit): T {
    func()
    return this
}
```

よりななをにする

```
fun doSomething() {
   return fluently { someOtherAction() }
}
```

letをするか、ヌルオブジェクトのをする

`let` Kotlinにそれはにびされたオブジェクトからバインディングロ―カルをします。

```
val str = "foo"
str.let {
    println(it) // it
}
```

これは`"foo"`、 `Unit`をします。

い*`let`*と*`also`*あなたからのをすことができるということである*`let`*ブロック。 *`also`*、ではに*`Unit`*を*reutrn*します。

なぜこれがなのですか `null`をすことができるメソッドをびすと、そのりが`null`でないにのみいくつかのコ―ドをしたい、 `let`をう`let also`できます`let`、のようにすること`also`できます

```
val str: String? = someFun()
str?.let {
    println(it)
}
```

このコ―ドは、 `str`が`null`でないにのみ`let`ブロックをし`null`。 `null ?` にしてください。

をしてオブジェクトをするか、メソッドをする

`apply`のには、のように`apply`ています。

　　されたブロックをびし`this`、そののようなをすと`this`は。

KDOCはそれほどではありませんが`apply`かになです。のでは`apply`れるを`this`あなたがばれるオブジェクトにバインドされている`apply`を。これにより、あるオブジェクトにしてのメソッドをびすがじたときに、そのオブジェクトをですときにいくつかのコードをすることができます。

```
File(dir).apply { mkdirs() }
```

これはこれをいているのとじです

```
fun makeDir(String path): File {
    val result = new File(path)
    result.mkdirs()
    return result
}
```

オンラインでをむ https://riptutorial.com/ja/kotlin/topic/2273/

31: メソッド

- fun TypeName.extensionNameparams、...{/ * body * /} //
- あなたはジェネリックスをってすることができます。<T> Any> TypeNameWithGenerics <T> .extensionNameparams、...{/ * body * /} // Generics
- myObj.extensionNameargs、...//びし

はにされます。つまり、するメソッドは、アクセスするのによってまります。にのがであるかはありません。じメソッドがにびされます。これは、メソッドをしても、にはタイプにメンバーがされないためです。

Examples

トップレベルの

のメソッドは、クラスにはまれていません。

```
fun IntArray.addTo(dest: IntArray) {
    for (i in 0 .. size - 1) {
        dest[i] += this[i]
    }
}
```

のメソッドは、`IntArrayIntArray`されています。メソッドがされているオブジェクト とばれるには、キーワード`this`をしてアクセスすることにしてください。

このはのようにびすことができます

```
val myArray = intArrayOf(1, 2, 3)
intArrayOf(4, 5, 6).addTo(myArray)
```

なとしがにされる

びされるメソッドは、アクセスされるのにづいてコンパイルにされます。にのがであるかはありません。じメソッドがにびされます。

```
open class Super

class Sub : Super()

fun Super.myExtension() = "Defined for Super"

fun Sub.myExtension() = "Defined for Sub"

fun callMyExtension(myVar: Super) {
    println(myVar.myExtension())
}
```

```
callMyExtension(Sub())
```

のでは、され"Defined for Super"のされたため、 myVarあるSuper。

がめるをレンダリングするためにくびるサンプル

がめるをするために、 Intまたは Longのがあれば、

```
fun Long.humanReadable(): String {
    if (this <= 0) return "0"
    val units = arrayOf("B", "KB", "MB", "GB", "TB", "EB")
    val digitGroups = (Math.log10(this.toDouble())/Math.log10(1024.0)).toInt();
    return DecimalFormat("#,##0.#").format(this/Math.pow(1024.0, digitGroups.toDouble())) + "
" + units[digitGroups];
}

fun Int.humanReadable(): String {
    return this.toLong().humanReadable()
}
```

ににされます

```
println(1999549L.humanReadable())
println(someInt.humanReadable())
```

サンプルJava 7+パスクラスの

メソッドのなは、のAPIをすることです。 Java 7+ Pathクラスに exist 、 notExists 、および deleteRecursivelyを existをにします。

```
fun Path.exists(): Boolean = Files.exists(this)
fun Path.notExists(): Boolean = !this.exists()
fun Path.deleteRecursively(): Boolean = this.toFile().deleteRecursively()
```

このでこれをびすことができます

```
val dir = Paths.get(dirName)
if (dir.exists()) dir.deleteRecursively()
```

をしてみやすくする

Kotlinではのようなコーードをくことができます

```
val x: Path = Paths.get("dirName").apply {
    if (Files.notExists(this)) throw IllegalStateException("The important file does not
exist")
}
```

しかし、 `apply`のは、あなたのとじくらいではありません。アクションのをしてよりなものにするために、のをすることが々になります。これはのかないようにすべきではありませんが、のようなになのために

```
infix inline fun <T> T.verifiedBy(verifyWith: (T) -> Unit): T {
    verifyWith(this)
    return this
}

infix inline fun <T: Any> T.verifiedWith(verifyWith: T.() -> Unit): T {
    this.verifyWith()
    return this
}
```

コーードをのようにくことができます

```
val x: Path = Paths.get("dirName") verifiedWith {
    if (Files.notExists(this)) throw IllegalStateException("The important file does not
exist")
}
```

これは、ラムダパラメーータでをするかを々にらせます。

`verifiedBy`パラメーータ`T`とじであることにしてください`T: Any?` nullなでもそのバーージョンのをできることをします。ただし、 `verifiedWith`はnullをできません。

ISO 8のをレンダリングするためのJava 8 Temporalクラスのサンプル

こので

```
fun Temporal.toIsoString(): String = DateTimeFormatter.ISO_INSTANT.format(this)
```

これでにのことができます

```
val dateAsString = someInstant.toIsoString()
```

コンパニオンオブジェクトへのの

クラスをするは、のたとえば、 `Something`クラスのstatic looking`fromString`するなどは、クラスにコンパニオンオブジェクトがあり、そのがコンパニオンオブジェクトでされているにのみします

```
class Something {
    companion object {}
}

class SomethingElse {
}

fun Something.Companion.fromString(s: String): Something = ...
```

```
fun SomethingElse.fromString(s: String): SomethingElse = ...

fun main(args: Array<String>) {
    Something.fromString("") //valid as extension function declared upon the
                             //companion object

    SomethingElse().fromString("") //valid, function invoked on instance not
                                   //statically

    SomethingElse.fromString("") //invalid
}
```

プロパティの

あなたがするのになプロパティをしたいとします。したがって、KotlinのKT-9686とKT- 13053でされているように、 レイジープロパティデリゲートをってをキャッシュし、のインスタンス `this` をすることはできません。ただし、ここにはのがあります 。

このでは、プロパティは`color`です。 これはな`colorCache`をしています。 `this`は`lazy`がでないため、 `this`とにできます

```
class KColor(val value: Int)

private val colorCache = mutableMapOf<KColor, Color>()

val KColor.color: Color
    get() = colorCache.getOrPut(this) { Color(value, true) }
```

なのためのコードからの

ビューをしたは、ビューのをすることができます。

Ankoによるオリジナルのアイデア

```
inline fun <reified T : View> View.find(id: Int): T = findViewById(id) as T
inline fun <reified T : View> Activity.find(id: Int): T = findViewById(id) as T
inline fun <reified T : View> Fragment.find(id: Int): T = view?.findViewById(id) as T
inline fun <reified T : View> RecyclerView.ViewHolder.find(id: Int): T =
itemView?.findViewById(id) as T

inline fun <reified T : View> View.findOptional(id: Int): T? = findViewById(id) as? T
inline fun <reified T : View> Activity.findOptional(id: Int): T? = findViewById(id) as? T
inline fun <reified T : View> Fragment.findOptional(id: Int): T? = view?.findViewById(id) as?
T
inline fun <reified T : View> RecyclerView.ViewHolder.findOptional(id: Int): T? =
itemView?.findViewById(id) as? T
```

```
val yourButton by lazy { find<Button>(R.id.yourButtonId) }
val yourText by lazy { find<TextView>(R.id.yourTextId) }
val yourEdittextOptional by lazy { findOptional<EditText>(R.id.yourOptionEdittextId) }
```

オンラインでメソッドをむ https://riptutorial.com/ja/kotlin/topic/613/メソッド

32:

Examples

の

Stringのは、インデックス`string[index]`によってアクセスできる`string[index]`です。

```
val str = "Hello, World!"
println(str[1]) // Prints e
```

は、forループでることができます。

```
for (c in str) {
    println(c)
}
```

リテラル

Kotlinには、2のリテラルがあります。

- エスケープ
- の

エスケープされたは、をエスケープしてします。エスケープはバックスラッシュでいます。のエスケープシーケンスがサポートされています `\t`、 `\b`、 `\n`、 `\r`、 `\'`、 `\"`、 `\\`および`\$` UnicodeエスケープシーケンスはFF00ます。

```
val s = "Hello, world!\n"
```

トリプルクォート`"""`でられたのは、エスケープをまず、やそののをむことができます

```
val text = """
    for (c in "foo")
        print(c)
"""
```

するはtrimMarginでりくことができます。

```
val text = """
    |Tell me and I forget.
    |Teach me and I remember.
    |Involve me and I learn.
    |(Benjamin Franklin)
    """.trimMargin()
```

のマージンプレフィックスはパイプです`|`これはtrimMarginのパラメータとしてできます。えば`trimMargin(">")`。

テンプレート

エスケープされたとののにテンプレートをめることができます。テンプレートはされるコードのであり、そのはにされます。ドル`$`まり、でされています

```
val i = 10
val s = "i = $i" // evaluates to "i = 10"
```

または、でまれたの

```
val s = "abc"
val str = "$s.length is ${s.length}" // evaluates to "abc.length is 3"
```

にリテラルドルをめるには、バックスラッシュをしてエスケープします。

```
val str = "\$foo" // evaluates to "$foo"
```

エスケープをサポートしていないのはです。のでは、のをしてドルをすことができます。

```
val price = """
${'$'}9.99
"""
```

Kotlinのは、`==`とされ、にであることがされます。

```
val str1 = "Hello, World!"
val str2 = "Hello," + " World!"
println(str1 == str2) // Prints true
```

は`===`でチェックされます。

```
val str1 = """
    |Hello, World!
    """.trimMargin()

val str2 = """
    #Hello, World!
    """.trimMargin("#")

val str3 = str1

println(str1 == str2) // Prints true
println(str1 === str2) // Prints false
println(str1 === str3) // Prints true
```

オンラインでをむ https://riptutorial.com/ja/kotlin/topic/8285/

33:

Javaの`switch`とはに、 `when`ステートメントにはフォールスルーがありません。 つまり、がした、フローはにり、 `break`はありません。 bahaviorsをのにしたいは、のをコンマでってくことができます

```
when (x) {
    "foo", "bar" -> println("either foo or bar")
    else -> println("didn't match anything")
}
```

Examples

のif

```
val str = "Hello!"
if (str.length == 0) {
    print("The string is empty!")
} else if (str.length > 5) {
    print("The string is short!")
} else {
    print("The string is long!")
}
```

else-branchesはのifではオプションです。

としてのIf

ifをにすることができます

```
val str = if (condition) "Condition met!" else "Condition not met!"
```

ことにしてください`else`-branchはオプションではありません`if` -statementがとしてされています。

これはとの`else if`ステートメントをむのでもできます。

```
val str = if (condition1){
    "Condition1 met!"
   } else if (condition2) {
    "Condition2 met!"
   } else {
    "Conditions not met!"
   }
```

> ヒントKotlinはのをすることができますが、のにをけるだけのは、 `val str: String =`をしてみやすくします。

if-else-ifチェーンのわりにwhen-statement

whenは、のelse ifブランチをつifのです。

```
when {
    str.length == 0 -> print("The string is empty!")
    str.length > 5  -> print("The string is short!")
    else            -> print("The string is long!")
}
```

*if-else-if*チェーンをってかれたじコード

```
if (str.length == 0) {
    print("The string is empty!")
} else if (str.length > 5) {
    print("The string is short!")
} else {
    print("The string is long!")
}
```

ifのとに、elseブランチはオプションであり、きなだけくのブランチまたはいくつかのブランチをできます。また、マルチラインブランチをすることもできます。

```
when {
    condition -> {
        doSomething()
        doSomeMore()
    }
    else -> doSomethingElse()
}
```

when-statementの

がえられると、 `when` -statementはをににします。は、 `==`をしてわれます。は、NULLチェックをし、 `equals`をしてオペランドをします。にするものがされます。

```
when (x) {
    "English" -> print("How are you?")
    "German" -> print("Wie geht es dir?")
    else -> print("I don't know that language yet :(")
}
```

whenステートメントにはさらになマッチングオプションがいくつかあります。

```
val names = listOf("John", "Sarah", "Tim", "Maggie")
when (x) {
    in names -> print("I know that name!")
    !in 1..10 -> print("Argument was not in the range from 1 to 10")
    is String -> print(x.length) // Due to smart casting, you can use String-functions here
}
```

when-としての

ifのように、whenをとしてすることもできます

```
val greeting = when (x) {
    "English" -> "How are you?"
    "German" -> "Wie geht es dir?"
    else -> "I don't know that language yet :("
}
print(greeting)
```

としてするには、whenはなものでなければなりません。つまり、elseブランチをつか、ブランチをつのですべてのをカバーします。

when-のステートメント

whenさせるためにすることができますenumを

```
enum class Day {
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
}

fun doOnDay(day: Day) {
    when(day) {
        Day.Sunday ->     // Do something
        Day.Monday, Day.Tuesday ->     // Do other thing
        Day.Wednesday ->  // ...
        Day.Thursday ->   // ...
        Day.Friday ->     // ...
        Day.Saturday ->   // ...
    }
}
```

2のケースライン MondayとTuedsay 2にられるように、2つのenumをみわせることもできます。

あなたのケースがでない、コンパイルはエラーをします。 elseをしてデフォルトのケースをできます

```
fun doOnDay(day: Day) {
    when(day) {
        Day.Monday ->     // Work
        Day.Tuesday ->    // Work hard
        Day.Wednesday ->  // ...
        Day.Thursday ->   //
        Day.Friday ->     //
        else ->           // Party on weekend
    }
}
```

じをしてうことができますが`if-then-else`、 `when`しているのを`enum`を、それがよりになります。

kotlin `enum`はこちらをごください

オンラインでをむ https://riptutorial.com/ja/kotlin/topic/2685/

34:

Examples

のマッチングのイディオム

のローカルの

「のファイル」テンプレートよりも、のスペースはなくなりますが、のスペースはきくなります。 whenがループにあるは、 "anonymous temporaries"テンプレートよりもします。この、のはループのにするがあります。

```
import kotlin.text.regex

var string = /* some string */

val regex1 = Regex( /* pattern */ )
val regex2 = Regex( /* pattern */ )
/* etc */

when {
    regex1.matches(string) -> /* do stuff */
    regex2.matches(string) -> /* do stuff */
    /* etc */
}
```

のファイルをする

「のローカル」テンプレートよりも、のスペースはなくなりますが、のスペースはきくなります。 expressionがループにあるwhenはしないでください。

```
import kotlin.text.regex

var string = /* some string */

when {
    Regex( /* pattern */ ).matches(string) -> /* do stuff */
    Regex( /* pattern */ ).matches(string) -> /* do stuff */
    /* etc */
}
```

パターンの

syntaxのwhen " - ful"をにエミュレートするがあります。これは、 whenのをよりにし、すべてのwhenEntry whenをりさなければならないことからじるのプログラマーいをするwhenEntryです。この

では、「のローカル」テンプレートまたは「のな」テンプレートのいずれかをビジターパターンとしてすることができます。

```
import kotlin.text.regex

var string = /* some string */

when (RegexWhenArgument(string)) {
    Regex( /* pattern */ ) -> /* do stuff */
    Regex( /* pattern */ ) -> /* do stuff */
    /* etc */
}
```

whenのラッパークラスの

```
class RegexWhenArgument (val whenArgument: CharSequence) {
    operator fun equals(whenEntry: Regex) = whenEntry.matches(whenArgument)
    override operator fun equals(whenEntry: Any?) = (whenArgument == whenEntry)
}
```

Kotlinのの

このでは、 Regexクラスのほとんどののい、 Regexににしたnullを Regex、およびのがパターンのきみとみみをにするをします。

RegExクラス

Kotlinでをするには、 Regex(pattern: String)クラスをし、そのオブジェクトで find(..)や replace(..)ようなをびすがあります。

inputにcまたはdがまれているにtrueをすRegexクラスをするの

```
val regex = Regex(pattern = "c|d")
val matched = regex.containsMatchIn(input = "abc")    // matched: true
```

すべてのRegexですることは、がpatternとinputのマッチングにづいていることです。のにはがなものもあれば、のみがなものもあります。このでされているcontainsMatchIn(..)は、がで、こののでします。

によるNullの

find(..)とmatchEntire(..)のMatchResult?オブジェクト。 ? Kotlinがnullをにするためには、MatchResultのMatchResultがです。

find(..)がnullをすとき、Kotlinが Regexからnullをにするをす

```
val matchResult =
    Regex("c|d").find("efg")           // matchResult: null
val a = matchResult?.value             // a: null
val b = matchResult?.value.orEmpty()   // b: ""
a?.toUpperCase()                       // Still needs question mark. => null
b.toUpperCase()                        // Accesses the function directly. => ""
```

orEmpty()をすると、 bをnullにすることはできません? bをびすと、はです。

あなたがnullのこのないににしないなら、KotlinはJavaのようにヌルですることをにします!!

```
a!!.toUpperCase()                      // => KotlinNullPointerException
```

パターンのの

Kotlinは、Java をってJava をしています。これにより、Javaでな、バックスラッシュのないなパターンをくことができます。のはでされます

```
"""\d{3}-\d{3}-\d{4}""" // raw Kotlin string
"\\d{3}-\\d{3}-\\d{4}"  // standard Java string
```

findinputCharSequence、startIndexInt MatchResult

inputはRegexオブジェクトのpatternとされます。 Matchresult?しますMatchresult? startIndexににするテキストをつオブジェクト。パターンがinputとしなかったはnull。 のはMatchResult?からされMatchResult?オブジェクトのvalueプロパティ。 startIndexパラメーターはで、は0です。

のをむからのなをするには

```
val phoneNumber :String? = Regex(pattern = """\d{3}-\d{3}-\d{4}""")
    .find(input = "phone: 123-456-7890, e..")?.value // phoneNumber: 123-456-7890
```

inputになが ない、phoneNumberはnullになりnull。

findAllinputCharSequence、startIndexIntシーケンス

patternするinputからすべてのをします。

でられたすべてのを、とのあるテキストからするには

```
val matchedResults = Regex(pattern = """\d+""").findAll(input = "ab12cd34ef")
val result = StringBuilder()
for (matchedText in matchedResults) {
    result.append(matchedText.value + " ")
}

println(result) // => 12 34
```

matchedResultsは、 MatchResultオブジェクトをつMatchResultです。なしのinputでは、 findAll(..)はのシーケンスをします。

matchEntireCharSequenceMatchResult

inputのすべてのがpatternとする、 inputとしいがされます。それのは、 nullがされます。

がの、をします。

```
val a = Regex("""\d+""").matchEntire("100")?.value             // a: 100
val b = Regex("""\d+""").matchEntire("100 dollars")?.value     // b: null
```

matchesinputCharSequenceBoolean

がパターンとするにをします。そうでなければFalse。

2つのにだけがまれているかどうかをテストします。

```
val regex = Regex(pattern = """\d+""")
regex.matches(input = "50")             // => true
regex.matches(input = "50 dollars")     // => false
```

containsMatchInCharSequenceブール

のがパターンとするにをします。そうでなければFalse。

2つのになくとも1つのがまれているかどうかをテストする

```
Regex("""\d+""").containsMatchIn("50 dollars")       // => true
Regex("""\d+""").containsMatchIn("Fifty dollars")    // => false
```

splitCharSequence、 limitIntリスト

すべてのマッチのないしいリストをします。

のないリストをすには

```
val a = Regex("""\d+""").split("ab12cd34ef")      // a: [ab, cd, ef]
val b = Regex("""\d+""").split("This is a test") // b: [This is a test]
```

のリストには1つのがあります。の`input`には3つのがあります。その、3つのからなるリストがられます。

CharSequence、String

`input`の`pattern`すべてのをにきえます。

のすべてのをx

```
val result = Regex("""\d+""").replace("ab12cd34ef", "x") // result: abxcdxef
```

オンラインでをむ https://riptutorial.com/ja/kotlin/topic/8364/

35:

Examples

アノテーションの

アノテーションは、メタデータをコードにするです。をするには、をクラスのにします。

```
annotation class Strippable
```

はメタアノテーションをつことができます

```
    @Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION,
AnnotationTarget.VALUE_PARAMETER, AnnotationTarget.EXPRESSION)
    annotation class Strippable
```

はのクラスとに、コンストラクタをつことができます。

```
annotation class Strippable(val importanceValue: Int)
```

しかし、のクラスとはなり、のタイプにされています

- Javaプリミティブ Int、Longなどにする。
-
- クラス Foo :: class
-
- そのの
- のタイプの

メタ

をするとき、のメタをしてメタをめることができます。

- `@Target` でできるクラス、、プロパティ、などのなをします。
- `@Retention` は、コンパイルされたクラスファイルにがされているかどうか、また、にリフレクションでできるかどうかをしますデフォルトではともtrueです。
- `@Repeatable` は、のにしてじアノテーションをすることをにします。
- `@MustBeDocumented` は、アノテーションがパブリックAPIのであることをし、されたAPIドキュメントにされているクラスまたはメソッドのシグネチャにめるがあります。

```
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION,
        AnnotationTarget.VALUE_PARAMETER, AnnotationTarget.EXPRESSION)
@Retention(AnnotationRetention.SOURCE)
```

```
@MustBeDocumented
annotation class Fancy
```

オンラインでをむ https://riptutorial.com/ja/kotlin/topic/4074/

36:

Examples

Kotlinのなは`Array<T>`されます。

のをするには、 `emptyArray<T>()`ファクトリをします。

```
val empty = emptyArray<String>()
```

されたサイズとでをするには、コンストラクタをします。

```
var strings = Array<String>(size = 5, init = { index -> "Item #$index" })
print(Arrays.toString(a)) // prints "[Item #0, Item #1, Item #2, Item #3, Item #4]"
print(a.size) // prints 5
```

には`get(index: Int): T`と`set(index: Int, value: T)`があります。

```
strings.set(2, "ChangedItem")
print(strings.get(2)) // prints "ChangedItem"

// You can use subscription as well:
strings[2] = "ChangedItem"
print(strings[2]) // prints "ChangedItem"
```

プリミティブの

これらのは 、ボクシングをけるために`Array<T>`からしませんが、じとメソッドをちます。

コトリン	ファクトリ	JVMタイプ
BooleanArray	booleanArrayOf(true, false)	boolean[]
ByteArray	byteArrayOf(1, 2, 3)	byte[]
CharArray	charArrayOf('a', 'b', 'c')	char[]
DoubleArray	doubleArrayOf(1.2, 5.0)	double[]
FloatArray	floatArrayOf(1.2, 5.0)	float[]
IntArray	intArrayOf(1, 2, 3)	int[]
LongArray	longArrayOf(1, 2, 3)	long[]
ShortArray	shortArrayOf(1, 2, 3)	short[]

`average()`は`Byte` 、 `Int` 、 `Long` 、 `Short` 、 `Double` 、 `Float`にしてされ、に`Double`します。

```
val doubles = doubleArrayOf(1.5, 3.0)
print(doubles.average()) // prints 2.25

val ints = intArrayOf(1, 4)
println(ints.average()) // prints 2.5
```

`component1()` 、 `component2()` 、 ... `component5()`はのをします

`getOrNull(index: Int)`は、indexがのはnullをし、そうでないはのをします。

`first()` 、 `last()`

`toHashSet()`はすべてのの`HashSet<T>`をします

`sortedArray()` 、 `sortedArrayDescending()`は、のソートされたをつしいをしてします

`sort()` 、 `sortDescending`をインプレースでソートする

`min()` 、 `max()`

の

Javaループとじループをしてをできますが、キーワードを`:`から`in`にするがあります。

```
val asc = Array(5, { i -> (i * i).toString() })
for(s : String in asc){
    println(s);
}
```

forループのデータをすることもできます。

```
val asc = Array(5, { i -> (i * i).toString() })
for(s in asc){
    println(s);
}
```

をする

```
val a = arrayOf(1, 2, 3) // creates an Array<Int> of size 3 containing [1, 2, 3].
```

クロージャをしてをする

```
val a = Array(3) { i -> i * 2 } // creates an Array<Int> of size 3 containing [0, 2, 4]
```

されていないをする

```
val a = arrayOfNulls<Int>(3) // creates an Array<Int?> of [null, null, null]
```

されるはにnullなをちます。 nullをしないのは、されていないではできません。

オンラインでをむ https://riptutorial.com/ja/kotlin/topic/5722/

37:

- しい ***Params*** = ...
- しい ***Params*** {...}
- しい ***Params*** タイプ {...}
- fun < タイプ > ***Params*** タイプ {...}
- inline fun ***Params*** タイプ {...}
- { ***ArgName*** ***ArgType*** - > ...}
- { ***ArgName*** - > ...}
- { ***ArgNames*** - > ...}
- { ***ArgName*** ***ArgType*** - > ...}

パラ メ―タ―

パラ メ―タ	
	の
Params	にえられたとの ***Name : Type***
タイプ	のりの
	ジェネリックプログラミングでされるパラメ―タずしもをすはない
ArgName	にえられたの
ArgType	***ArgName***の
ArgNames	コンマでられたArgNameのリスト

Examples

のをる

「ラムダ」にられるように、はのをパラメ―タとしてることができます。のをつをするためにな""はのとおりです

```
# Takes no parameters and returns anything
() -> Any?

# Takes a string and an integer and returns ReturnType
(arg1: String, arg2: Int) -> ReturnType
```

たとえば、なの`() -> Any?`できます`() -> Any?`ラムダを2するをする

```
fun twice(x: () -> Any?) {
    x(); x();
}

fun main() {
    twice {
        println("Foo")
    } # => Foo
      # => Foo
}
```

ラムダ

ラムダは、、びしのにパラメーータとしてするようにされるです。がな、これらはのにかれ-らは{}でをむことでされています-> 。

```
{ name: String ->
    "Your name is $name" //This is returned
}
```

ラムダののののステートメントはにりです。

コンパイラがをできるにラムダをくと、はオプションです。

の

```
{ argumentOne:String, argumentTwo:String ->
    "$argumentOne - $argumentTwo"
}
```

ラムダが1つのしかとしないは、リストをしitわりに1つのをすることができます。

```
{ "Your name is $it" }
```

へののがラムダである、びしからかっこをにすることができます。

```
# These are identical
listOf(1, 2, 3, 4).map { it + 2 }
listOf(1, 2, 3, 4).map({ it + 2 })
```

ののに::けることで、をにびさずにをすることができます。これは、のをパラメーータとしてけれるにすことができます。

```
fun addTwo(x: Int) = x + 2
listOf(1, 2, 3, 4).map(::addTwo) # => [3, 4, 5, 6]
```

せずにをにされます(ParamTypeA, ParamTypeB, ...) -> ReturnType ParamTypeA 、 ParamTypeB ... ReturnType1はのりのタイプである`パラメーータのタイプであり、。

```
fun foo(p0: Foo0, p1: Foo1, p2: Foo2): Bar {
    //...
}
println(::foo::class.java.genericInterfaces[0])
// kotlin.jvm.functions.Function3<Foo0, Foo1, Foo2, Bar>
// Human readable type: (Foo0, Foo1, Foo2) -> Bar
```

レシ―バであろうとメンバであろうとをつは、なったをっています。ダブルコロンのにレシ―バのタイプをするがあります

```
class Foo
fun Foo.foo(p0: Foo0, p1: Foo1, p2: Foo2): Bar {
    //...
}
val ref = Foo::foo
println(ref::class.java.genericInterfaces[0])
// kotlin.jvm.functions.Function4<Foo, Foo0, Foo1, Foo2, Bar>
// Human readable type: (Foo, Foo0, Foo1, Foo2) -> Bar
// takes 4 parameters, with receiver as first and actual parameters following, in their order

// this function can't be called like an extension function, though
val ref = Foo::foo
Foo().ref(Foo0(), Foo1(), Foo2()) // compile error

class Bar {
    fun bar()
}
print(Bar::bar) // works on member functions, too.
```

しかし、のがオブジェクトである、はそのようなのインスタンスの1つだけであるため、パラメ―タリストからされます。

```
object Foo
fun Foo.foo(p0: Foo0, p1: Foo1, p2: Foo2): Bar {
    //...
}
val ref = Foo::foo
println(ref::class.java.genericInterfaces[0])
// kotlin.jvm.functions.Function3<Foo0, Foo1, Foo2, Bar>
// Human readable type: (Foo0, Foo1, Foo2) -> Bar
// takes 3 parameters, receiver not needed

object Bar {
    fun bar()
}
print(Bar::bar) // works on member functions, too.
```

kotlin 1.1、リファレンスはにバインドすることもできます。これは、バウンドリファレンスとばれます 。

1.1.0

```
fun makeList(last: String?): List<String> {
    val list = mutableListOf("a", "b", "c")
    last?.let(list::add)
    return list
```

```
}
```

このは、がどのようにするかをすためにのみえられていることにしてください。のすべてのではいです。

しかし、なケースがあります。メンバとしてされたはできません。

```
class Foo
class Bar {
    fun Foo.foo() {}
    val ref = Foo::foo // compile error
}
```

は`fun`キーワードをしてされ、そのにとパラメータがきます。のりのをすることもできます。は`Unit`です。のは`{}`まれています。りのが`Unit`の、は、のすべてのブランチにしてreturnをするがあります。

```
fun sayMyName(name: String): String {
    return "Your name is $name"
}
```

じものの

```
fun sayMyName(name: String): String = "Your name is $name"
```

また、はできるのですることができます

```
fun sayMyName(name: String) = "Your name is $name"
```

にが1つだけまれているは、のように、をしてわりにequalsをできます。のはにされます。

```
fun sayMyName(name: String): String = "Your name is $name"
```

インライン

は、 `inline`をして`inline`ですることができます。この、びされるのではなく、Cのマクロのようにします。コンパイルにのコードにきえられます。これは、にラムダがパラメータとしてされるに、パフォーマンスのをもたらすことがあります。

```
inline fun sayMyName(name: String) = "Your name is $name"
```

Cマクロとのいの1つは、インラインがびされたスコープにインラインがアクセスできないことです。

```
inline fun sayMyName() = "Your name is $name"

fun main() {
    val name = "Foo"
```

```
    sayMyName() # => Unresolved reference: name
}
```

Kotlinは `+`や`*` とをつみののセットをすることをにします。をするために、するのをつメンバまたはをします。オーバーロードする`operator`に`operatoroperator`けるがある

```
data class IntListWrapper (val wrapped: List<Int>) {
    operator fun get(position: Int): Int = wrapped[position]
}

val a = IntListWrapper(listOf(1, 2, 3))
a[1] // == 2
```

よりくのがここにあります

オンラインでをむ https://riptutorial.com/ja/kotlin/topic/1280/

38: ののパラメータ

- **Vararg**キーワード `vararg`はメソッドでされ、のパラメータがけれられることをします。
- スプレッド `*`のパラメータにをするびしでされるののアスタリスク `*`。

Examples

varargキーワードの

`vararg`キーワードをしてをします。

```
fun printNumbers(vararg numbers: Int) {
    for (number in numbers) {
        println(number)
    }
}
```

これで、なのくのパラメータをにすことができます。

```
printNumbers(0, 1)                // Prints "0" "1"
printNumbers(10, 20, 30, 500)     // Prints "10" "20" "30" "500"
```

Varargパラメータは、パラメータリストののパラメータでなければなりません。

スプレッドを**vararg**にす

は、スプレッド `*`をしてvarargにすことができます。

のがするとします...

```
fun printNumbers(vararg numbers: Int) {
    for (number in numbers) {
        println(number)
    }
}
```

あなたはのようにをすことができます...

```
val numbers = intArrayOf(1, 2, 3)
printNumbers(*numbers)

// This is the same as passing in (1, 2, 3)
```

スプレッドは、パラメータのでることもできます...

```
val numbers = intArrayOf(1, 2, 3)
printNumbers(10, 20, *numbers, 30, 40)
```

```
// This is the same as passing in (10, 20, 1, 2, 3, 30, 40)
```

オンラインでののパラメータをむ https://riptutorial.com/ja/kotlin/topic/5835/ののパラメータ

クレジット

S. No		Contributors
1	Kotlinをいめる	babedev, Community, cyberscientist, ganesshkumar, Ihor Kucherenko, Jayson Minard, mnoronha, neworld, Parker Hoyes, Ruckus T-Boom, Sach, Sean Reilly, Sheigutn, Simón Oroño, UnKnown, Urko Pineda
2	DSLビルディング	Dmitriy L, ice1000
3	Java 8ストリームにするもの	Brad, Gerson, Jayson Minard, Piero Divasto, Sam
4	JavaのためのKotlin	Thorsten Schleinzer
5	JUnit	jenglert
6	Kotlin Android	Jemo Mgebrishvili, Ritave
7	Kotlinの RecyclerView	Mohit Suthar
8	Kotlinのループ	Ben Leggiero, JaseAnderson, mayojava, razzledazzle, Robin
9	Kotlinビルドの	Aaron Christiansen, elect, madhead
10	Nullセーフティ	KeksArmee, Kirill Rakhman, piotrek1543, razzledazzle, Robin, SerCe, Spidfire, technerd, Thorsten Schleinzer
11	インターフェイス	Divya, Jan Vladimir Mostert, Jayson Minard, Ritave, Robin
12	クラス	Sam
13	クラス	byxor, KeksArmee, piotrek1543, Slav
14	コトリンにログインする	Konrad Jamrozik, olivierlemasle, oshai
15	コトリンの	Shinoo Goyal
16	コトリン	Grigory Konushev, Spidfire
17	コルーチン	Jemo Mgebrishvili
18	コレクション	Ascension

19	ジェネリックス	hotkey, Jayson Minard, KeksArmee
20	シングルトンオブジェクト	Divya, glee8e
21	タイプエイリアス	Kevin Robatel
22	タイプセーフビルダ	Slav
23	レンジ	Nihal Saxena
24		Brad Larson, jereksel, Sapan Zaveri
25		David Soroko, Kirill Rakhman, SerCe
26		atok, Kirill Rakhman, madhead, Ritave, Sup
27		Avijit Karmakar
28	なラムダ	memoizr, Rich Kuzsma
29	されたプロパティ	Sam, Seaskyways
30		Aaron Christiansen, Adam Arold, Brad Larson, Héctor, Jayson Minard, Konrad Jamrozik, madhead, mayojava, razzledazzle, Sapan Zaveri, Serge Nikitin, yole
31	メソッド	Dávid Tímár, Jayson Minard, Kevin Robatel, Konrad Jamrozik, olivierlemasle, Parker Hoyes, razzledazzle
32		Januson, Sam
33		Abdullah, Alex Facciorusso, jpmcosta, Kirill Rakhman, Robin, Spidfire
34		Espen, Travis
35		Brad Larson, Caelum, Héctor, Mood, piotrek1543, Sapan Zaveri
36		egor.zhdan, Sam, UnKnown
37		Aaron Christiansen, baha, Caelum, glee8e, Jayson Minard, KeksArmee, madhead, Spidfire
38	ののパラメータ	byxor, piotrek1543, Sam