無料電子ブック

学習

Scala Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#scala

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

It is an unofficial and free Scala 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 Scala Language.

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: Scalaをいめる

Scalaは、なプログラミングパターンをでエレガントでタイプセーフなでするためにされた、のマルチパラダイムプログラミングです。 オブジェクトとのをスムーズにします。

ほとんどのでは、Scalaのインストールがです。 これはScalaのインストールページです 。これは「Scalaの」のです。 scalafiddle.netは、Webでさなコードをするためのれたリソースです。

バージョン

バージョン	
2.10.1	2013-03-13
2.10.2	2013-06-06
2.10.3	2013101
2.10.4	2014-03-24
2.10.5	2015-03-05
2.10.6	2015-09-18
2.11.0	2014-04-21
2.11.1	2014-05-21
2.11.2	2014-07-24
2.11.4	2014-10-30
2.11.5	2014-01-14
2.11.6	2015-03-05
2.11.7	2015-06-23
2.11.8	2016-03-08
2.11.11	2017-04-19
2.12.0	20161103
2.12.1	2016-12-06
2.12.2	2017-04-19

Examples

「メイン」メソッドのによるHello World

このコードを`HelloWorld.scala`というのファイルにします。

```
object Hello {
  def main(args: Array[String]): Unit = {
    println("Hello World!")
  }
}
```

ライブデモ

JVMによってなバイトコードにコンパイルするには

```
$ scalac HelloWorld.scala
```

それをするには

```
$ scala Hello
```

Scalaランタイムがプログラムをロードすると、`main`メソッドをつ`Hello`というのオブジェクトがされます。`main`はプログラムエントリポイントであり、されます。

Scalaには、Javaとはって、オブジェクトやクラスのファイルのにするはありません。わりに、`Hello`というパラメータがコマンド`scala Hello`されます`scala Hello`はする`main`メソッドをむオブジェクトをします。じ`.scala`ファイルにmainメソッドをつのオブジェクトをつことはにです。

`args`には、もしあれば、プログラムにえられたコマンドラインがまれます。たとえば、のようにプログラムをすることができます。

```
object HelloWorld {
  def main(args: Array[String]): Unit = {
    println("Hello World!")
    for {
      arg <- args
    } println(s"Arg=$arg")
  }
}
```

それをコンパイルする

```
$ scalac HelloWorld.scala
```

そしてそれをしてください

```
$ scala HelloWorld 1 2 3
```

```
Hello World!
Arg=1
Arg=2
Arg=3
```

アプリをしてHello World

```
object HelloWorld extends App {
  println("Hello, world!")
}
```

ライブデモ

App をすることで、mainメソッドのをけることができます。 HelloWorldオブジェクトのが「メインメソッド」としてわれます。

2.11.0

> のドキュメントによれば 、 AppはDelayed Initializationというをしています。これは、メインメソッドがびされたにオブジェクトフィールドがされることをします。

2.11.0

> のドキュメントによれば 、 AppはDelayed Initializationというをしています。これは、メインメソッドがびされたにオブジェクトフィールドがされることをします。
>
> DelayedInitはなでになりましたが、なケースとしてAppではききサポートされています。がされされるまで、サポートはされます。

Appするときにコマンドラインにアクセスするには、 this.argsしthis.args。

```
object HelloWorld extends App {
  println("Hello World!")
  for {
    arg <- this.args
  } println(s"Arg=$arg")
}
```

するときはApp 、ののようにされるmain、オーバーライドするはありませんmain。

スクリプトとしてのHello World

Scalaはスクリプトとしてできます。デモをうには、ののHelloWorld.scalaをします。

```
println("Hello")
```

コマンドラインインタプリタでします $はコマンドラインプロンプトです。

```
$ scala HelloWorld.scala
Hello
```

`.scala` に`scala HelloWorld`としたなどをすると、ランナーはコンパイルしてスクリプトをするのではなく、バイトコードでコンパイルされた`.class`ファイルをします。

scalaがスクリプトとしてされている、パッケージはできません。

`bash`などのシェルをするオペレーティングシステムでは、Scalaスクリプトは 'シェルプリアンブル'をしてできます。 `HelloWorld.sh`というのファイルをし、のをコンテンツとしてします。

```
#!/bin/sh
exec scala "$0" "$@"
!#
println("Hello")
```

`#!`の`#! !#`は「シェルプリアンブル」であり、bashスクリプトとしてされます。 りはScalaです。

のファイルをしたら、そのファイルに「」をえるがあります。シェルでこれをうことができます

```
$ chmod a+x HelloWorld.sh
```

これはすべてのにをえることにしてください chmodをんでもっとのユーザのためにするをんでください。

これでスクリプトをできます

```
$ ./HelloWorld.sh
```

Scala REPLの

パラメータなしでで`scala`をすると、 REPL Read-Eval-Print Loopインタプリタがきます。

```
nford:~ $ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_66).
Type in expressions for evaluation. Or try :help.

scala>
```

REPLをすると、ワークシートでScalaをできます。コンテキストはされ、プログラムをすることなくでコマンドをすことができます。えば、 `val poem = "As halcyons we shall be"`とすると、のようになります。

```
scala> val poem = "As halcyons we shall be"
poem: String = As halcyons we shall be
```

、たちは`val`をすることができます

```
scala> print(poem)
As halcyons we shall be
```

`val`はであり、きできないことにしてください。

```
scala> poem = "Brooding on the open sea"
<console>:12: error: reassignment to val
       poem = "Brooding on the open sea"
```

しかし、REPLでは`val` をすることができますじスコープでされた、のScalaプログラムでエラーがします。

```
scala> val poem = "Brooding on the open sea"
poem: String = Brooding on the open sea
```

REPLセッションのりのでは、このしくされたは、にされたをシャドウします。 REPLは、オブジェクトやのコードがどのようにするかをくするのにです。 Scalaのすべてのができます、クラス、メソッドなどをすることができます。

スカラクイックシート

	コード
のintをする	`val x = 3`
intをする	`var x = 3`
なでのをする	`val x: Int = 27`
れてされたをりてる	`lazy val y = print("Sleeping in.")`
をにバインドする	`val f = (x: Int) => x * x`
なをつにをバインドする	`val f: Int => Int = (x: Int) => x * x`
メソッドをする	`def f(x: Int) = x * x`
なをつメソッドをする	`def f(x: Int): Int = x * x`
クラスをする	`class Hopper(someParam: Int) { ... }`
オブジェクトをする	`object Hopper(someParam: Int) { ... }`
の	`trait Grace { ... }`
シーケンスののをする	`Seq(1,2,3).head`
スイッチ	`val result = if(x > 0) "Positive!"`

	コード
のものをくシーケンスのすべてのをする	`Seq(1,2,3).tail`
リストをループする	`for { x <- Seq(1,2,3) } print(x)`

れループ	`for {`
	`x <- Seq(1,2,3)`
	`y <- Seq(4,5,6)`
	`} print(x + ":" + y)`

リストにしてexecute	`List(1,2,3).foreach { println }`
にする	`print("Ada Lovelace")`
リストをでソートする	`List('b','c','a').sorted`

オンラインでScalaをいめるをむ https://riptutorial.com/ja/scala/topic/216/scalaをいめる

2: Gradleをってする

Examples

1. ののSCALA_PROJECT/build.gradleというのファイルをします。

```
group 'scala_gradle'
version '1.0-SNAPSHOT'

apply plugin: 'scala'

repositories {
    jcenter()
    mavenCentral()
    maven {
        url "https://repo.typesafe.com/typesafe/maven-releases"
    }
}

dependencies {
    compile group: 'org.scala-lang', name: 'scala-library', version: '2.10.6'
}


task  "create-dirs" << {
    sourceSets*.scala.srcDirs*.each { it.mkdirs() }
    sourceSets*.resources.srcDirs*.each { it.mkdirs() }
}
```

2. gradle tasksして、なgradle tasksをします。

3. gradle create-dirsをして、 src/scala, src/resourcesディレクトリをします。

4. gradle buildをして、プロジェクトをビルドし、をダウンロードします。

のGradle Scalaプラグインをする

Basic Setupのをした、Scala Gradleプロジェクトごとにそのほとんどをりしているかもしれません。ボイラープレートコードのようない...

Gradleがするプラグインをするわりに、のプラグインをにする、すべてののビルドロジックをするのScalaプラグインをすることができます。

このでは、のビルドロジックをなGradleプラグインにします。

いなことに、Gradleでは、 ドキュメントにされているように、Gradle APIのけをりてカスタムプラグインをにくことができます 。として、ScalaまたはJavaをすることができます。しかし、ドキュメントをしてつかるサンプルのほとんどはGroovyでかれています。よりくのコードサンプルがなや、Scalaプラグインのにあるものをしたい、たとえば、gradle github repoをチェックするこ

とができます。

プラグインの

カスタムプラグインは、プロジェクトにするとのをします。

- `scalaVersion`プロパティオブジェクト。オーバーライドなデフォルトプロパティが2つあります。
  - メジャー= "2.12"
  - マイナー= "0"
- `withScalaVersion`のにされるは、バイナリをするためにScalaのメジャーバージョンがされますSBTの`%%`オペレータがベルをらすかもしれないが、そうでないはくここにむに
- のとまったくじように、なディレクトリツリーをする`createDirs`タスク

ガイドライン

1. しいgradleプロジェクトをし、 `build.gradle`をします

```
apply plugin: 'scala'
apply plugin: 'maven'

repositories {
    mavenLocal()
    mavenCentral()
}

dependencies {
    compile gradleApi()
    compile "org.scala-lang:scala-library:2.12.0"
}
```

- プラグインのはScalaでかれているので、Gradle Scala Pluginがです
- のプロジェクトのプラグインをするには、Gradle Maven Pluginをします。これにより、プロジェクトjarをMavenローカルリポジトリに`install`ためにされる`install`タスクがされます
- `compile gradleApi()`を`compile gradleApi() gradle-api-<gradle_version>.jar`がクラスパスにされます

2. プラグインのにしいScalaクラスをする

```
package com.btesila.gradle.plugins

import org.gradle.api.{Plugin, Project}

class ScalaCustomPlugin extends Plugin[Project] {
    override def apply(project: Project): Unit = {
        project.getPlugins.apply("scala")
    }
}
```

- プラグインをするには、にProjectのPluginをし、 applyメソッドをオーバーライドしてください
- applyメソッドでは、プラグインがされているProjectインスタンスへのアクセスがあり、それをしてビルドロジックをすることができます
- このプラグインは、のGradle Scala Pluginをにしています。

3. scalaVersionオブジェクトプロパティをする

まず、 ScalaVersionクラスをします。 ScalaVersionクラスは、2つのバージョンプロパティ

```
class ScalaVersion {
  var major: String = "2.12"
  var minor: String = "0"
}
```

Gradleプラグインにするれたの1つは、のプロパティをにまたはきできるということです。プラグインは、こののユーザーを、gradle ProjectインスタンスにされたExtensionContainerでけります。については、 これをチェックしてください。
applyメソッドにをすることによって、にこれをします。

- プロジェクトにscalaVersionプロパティがされていないは、デフォルトをします
- さもなければ、 ScalaVersionインスタンスとしてのインスタンスをScalaVersion 、さらにそれをします

```
var scalaVersion = new ScalaVersion
if (!project.getExtensions.getExtraProperties.has("scalaVersion"))
   project.getExtensions.getExtraProperties.set("scalaVersion", scalaVersion)
else
   scalaVersion =
project.getExtensions.getExtraProperties.get("scalaVersion").asInstanceOf[ScalaVersion]
```

これは、プラグインをするプロジェクトのビルドファイルにをきむのとじです

```
ext {
    scalaVersion.major = "2.12"
    scalaVersion.minor = "0"

}
```

4. scalaVersionをして、 scala-langライブラリをプロジェクトのにしscalaVersion

```
project.getDependencies.add("compile", s"org.scala-lang:scala-
library:${scalaVersion.major}.${scalaVersion.minor}")
```

これは、プラグインをするプロジェクトのビルドファイルにをきむのとじです

```
compile "org.scala-lang:scala-library:2.12.0"
```

5. `withScalaVersion`をする

```
val withScalaVersion = (lib: String) => {
    val libComp = lib.split(":")
    libComp.update(1, s"${libComp(1)}_${scalaVersion.major}")
    libComp.mkString(":")
}
project.getExtensions.getExtraProperties.set("withScalaVersion", withScalaVersion)
```

6. に、 `createDirs`タスクをしてプロジェクトにします
   `DefaultTask`をしてGradleタスクをし`DefaultTask`。

```
class CreateDirs extends DefaultTask {
  @TaskAction
  def createDirs(): Unit = {
    val sourceSetContainer =
this.getProject.getConvention.getPlugin(classOf[JavaPluginConvention]).getSourceSets

    sourceSetContainer forEach { sourceSet =>
      sourceSet.getAllSource.getSrcDirs.forEach(file => if (!file.getName.contains("java"))
file.mkdirs())
    }
  }
}
```

`SourceSetContainer`は、プロジェクトにするすべてのソースディレクトリにするがあります。
Gradle Scala Pluginのは、 プラグインのドキュメントにあるように、なソースセットをJavaにすることです。

これを`apply`メソッドにして、プロジェクトに`createDir`タスクをし`apply`。

```
project.getTasks.create("createDirs", classOf[CreateDirs])
```

、 `ScalaCustomPlugin`クラスはのようになります

```
class ScalaCustomPlugin extends Plugin[Project] {
  override def apply(project: Project): Unit = {
    project.getPlugins.apply("scala")

    var scalaVersion = new ScalaVersion
    if (!project.getExtensions.getExtraProperties.has("scalaVersion"))
      project.getExtensions.getExtraProperties.set("scalaVersion", scalaVersion)
    else
      scalaVersion =
project.getExtensions.getExtraProperties.get("scalaVersion").asInstanceOf[ScalaVersion]

    project.getDependencies.add("compile", s"org.scala-lang:scala-
library:${scalaVersion.major}.${scalaVersion.minor}")

    val withScalaVersion = (lib: String) => {
      val libComp = lib.split(":")
      libComp.update(1, s"${libComp(1)}_${scalaVersion.major}")
      libComp.mkString(":")
```

```
    }
    project.getExtensions.getExtraProperties.set("withScalaVersion", withScalaVersion)

    project.getTasks.create("createDirs", classOf[CreateDirs])
  }
}
```

プラグインプロジェクトをロ―カルの**Maven**リポジトリにインスト―ルする

これは、 `gradle installgradle install`ことでとてもにされます
インスト―ルをするには、ロ―カルのリポジトリディレクトリは`~/.m2/repository`

Gradleはしいプラグインをどのようにつけますか

Gradleプラグインには、 `apply`ステ―トメントでされる`id`あり`apply`。えば、をビルドファイルにきむことで、Gradleのトリガにされ、id `scala`プラグインをつけてします。

```
apply plugin: 'scala'
```

じように、しいプラグインをのようにしたいといます。

```
apply plugin: "com.btesila.scala.plugin"
```

たちのプラグインは`com.btesila.scala.plugin` idをつことに`com.btesila.scala.plugin`ます。

このidをするには、のファイルをします。
src / main / resources / META-INF / gradle-plugin / com.btesil.scala.plugin.properties

```
implementation-class=com.btesila.gradle.plugins.ScalaCustomPlugin
```

その、`gradle install`します。

プラグインの

1. しいのGradleプロジェクトをし、をビルドファイルにします

```
buildscript {
    repositories {
        mavenLocal()
        mavenCentral()
    }

    dependencies {
        //modify this path to match the installed plugin project in your local repository
        classpath 'com.btesila:working-with-gradle:1.0-SNAPSHOT'
    }
}
```

```
repositories {
    mavenLocal()
    mavenCentral()
}


apply plugin: "com.btesila.scala.plugin"
```

2. `gradle createDirsgradle createDirs` - すべてのソースディレクトリがされるはずです
3. これをビルドファイルにしてスカラのバージョンをきする

```
ext {
    scalaVersion.major = "2.11"
    scalaVersion.minor = "8"

}
println(project.ext.scalaVersion.major)
println(project.ext.scalaVersion.minor)
```

4. Scalaバージョンとバイナリのあるライブラリをする

```
dependencies {
    compile withScalaVersion("com.typesafe.scala-logging:scala-logging:3.5.0")
}
```

それでおしまいいプラグインをりすことなく、すべてのプロジェクトでこのプラグインをできるようになりました。

オンラインでGradleをってするをむ https://riptutorial.com/ja/scala/topic/3304/gradleをってする

3: Javaの

Examples

スカラコレクションをJavaコレクションにするも

コレクションをJavaメソッドにすがあるは、のようにします。

```
import scala.collection.JavaConverters._

val scalaList = List(1, 2, 3)
JavaLibrary.process(scalaList.asJava)
```

JavaコードがJavaコレクションをすは、のでScalaコレクションにできます。

```
import scala.collection.JavaConverters._

val javaCollection = JavaLibrary.getList
val scalaCollection = javaCollection.asScala
```

これらはデコレータであることにしてください。そのため、ScalaやJavaのコレクションインタフェースでとなるコレクションをにラップするだけです。したがって、 `.asJava`と`.asScala`のびしはコレクションをコピーしません。

は、のJVMであり、それらはとしてわれ、なコンストラクタとなをっています。 `new`キーワードなしでそれらをしてください。

```
 val a = Array("element")
```

さてをつ`a Array[String]`。

```
 val acs: Array[CharSequence] = a
 //Error: type mismatch;  found   : Array[String]  required: Array[CharSequence]
```

`String`は`CharSequence`に`CharSequence`、 `Array[String]`は`Array[CharSequence]`できません。

`TraversableLike ArrayOps`へのなのおかげで、のコレクションとに`Array`をできます。

```
 val b: Array[Int] = a.map(_.length)
```

ほとんどのScalaコレクション `TraversableOnce` には、の`ClassTag`をしてをする`toArray`メソッドがあります。

```
 List(0).toArray
 //> res1: Array[Int] = Array(0)
```

これにより、 `TraversableOnce` をScalaコードでいやすくし、をとするJavaコードにすことがになります。

スカラとJavaの

Scalaは、JavaConvertersオブジェクトのすべてのなコレクションのなをします。

のはです。

スカラ	Javaタイプ
イテレータ	java.util.Iterator
イテレータ	java.util.Enumeration
イテレータ	java.util.Iterable
イテレータ	java.util.Collection
mutable.Buffer	java.util.List
mutable.Set	java.util.Set
mutable.Map	java.util.Map
mutable.ConcurrentMap	java.util.concurrent.ConcurrentMap

ののScalaコレクションもJavaにできますが、のScalaにされません。

スカラ	Javaタイプ
Seq	java.util.List
mutable.Seq	java.util.List
セット	java.util.Set
	java.util.Map

リファレンス

JavaコレクションとScalaコレクションの

スカラのインタフェース - scala-java8-compat

ScalaのJava 8キット。

ほとんどのはReadmeからコピーされています

scala.FunctionNと**java.util.function**のの

```
import java.util.function._
import scala.compat.java8.FunctionConverters._

val foo: Int => Boolean = i => i > 7
def testBig(ip: IntPredicate) = ip.test(9)
println(testBig(foo.asJava))  // Prints true

val bar = new UnaryOperator[String]{ def apply(s: String) = s.reverse }
List("cod", "herring").map(bar.asScala)    // List("doc", "gnirrih")

def testA[A](p: Predicate[A])(a: A) = p.test(a)
println(testA(asJavaPredicate(foo))(4))  // Prints false
```

scala.Optionクラスと**java.util**クラスののコンバーターオプション、**OptionalDouble**、OptionalInt、および**OptionalLong**。

```
import scala.compat.java8.OptionConverters._

    class Test {
      val o = Option(2.7)
      val oj = o.asJava        // Optional[Double]
      val ojd = o.asPrimitive  // OptionalDouble
      val ojds = ojd.asScala   // Option(2.7) again
    }
```

Scalaコレクションから**Java 8**ストリームへの

```
import java.util.stream.IntStream

import scala.compat.java8.StreamConverters._
import scala.compat.java8.collectionImpl.{Accumulator, LongAccumulator}


  val m = collection.immutable.HashMap("fish" -> 2, "bird" -> 4)
  val parStream: IntStream = m.parValueStream
  val s: Int = parStream.sum
  // 6, potientially computed in parallel
  val t: List[String] = m.seqKeyStream.toScala[List]
  // List("fish", "bird")
  val a: Accumulator[(String, Int)] = m.accumulate // Accumulator[(String, Int)]

  val n = a.stepper.fold(0)(_ + _._1.length) +
    a.parStream.count // 8 + 2 = 10

  val b: LongAccumulator = java.util.Arrays.stream(Array(2L, 3L, 4L)).accumulate
  // LongAccumulator
  val l: List[Long] = b.to[List] // List(2L, 3L, 4L)
```

オンラインでJavaのをむ https://riptutorial.com/ja/scala/topic/2441/javaの

4: JSON

Examples

JSON with spray-json

spray-jsonはJSONでにできます。なフォーマットをすると、すべてが「で」こります。

SBTでライブラリをできるようにする

SBTがするライブラリのをつ`spray-json`をするには

```
libraryDependencies += "io.spray" %% "spray-json" % "1.3.2"
```

のパラメータであるバージョン `1.3.2` は、プロジェクトによってなるがあります。

`spray-json`ライブラリはrepo.spray.ioでホストされています 。

ライブラリをインポートする

```
import spray.json._
import DefaultJsonProtocol._
```

デフォルトのJSONプロトコル`DefaultJsonProtocol`は、すべてのタイプのフォーマットがまれています。カスタム·タイプにJSONをするには、フォーマットなBuilderをするか、にをいてください。

JSONをむ

```
// generates an intermediate JSON representation (abstract syntax tree)
val res = """{ "foo": "bar" }""".parseJson // JsValue = {"foo":"bar"}

res.convertTo[Map[String, String]] // Map(foo -> bar)
```

JSONをく

```
val values = List("a", "b", "c")
values.toJson.prettyPrint // ["a", "b", "c"]
```

DSL

DSLはサポートされていません。

ケースクラスへのみき

のは、ケース･クラス･オブジェクトをJSONにシリアルするをしています。

```
case class Address(street: String, city: String)
case class Person(name: String, address: Address)

// create the formats and provide them implicitly
implicit val addressFormat = jsonFormat2(Address)
implicit val personFormat = jsonFormat2(Person)

// serialize a Person
Person("Fred", Address("Awesome Street 9", "SuperCity"))
val fredJsonString = fred.toJson.prettyPrint
```

これにより、のJSONがされます。

```
{
  "name": "Fred",
  "address": {
    "street": "Awesome Street 9",
    "city": "SuperCity"
  }
}
```

に、JSONをシリアルしてオブジェクトにすことができます。

```
val personRead = fredJsonString.parseJson.convertTo[Person]
//Person(Fred,Address(Awesome Street 9,SuperCity))
```

カスタムフォーマット

のながなは、 カスタム`JsonFormat`ます。たとえば、Scalaでフィールドがjsonとなる。あるいは、なるコンクリートがについてインスタンスされている。

```
implicit object BetterPersonFormat extends JsonFormat[Person] {
    // deserialization code
    override def read(json: JsValue): Person = {
        val fields = json.asJsObject("Person object expected").fields
        Person(
            name = fields("name").convertTo[String],
            address = fields("home").convertTo[Address]
        )
    }
```

```
    // serialization code
    override def write(person: Person): JsValue = JsObject(
        "name" -> person.name.toJson,
        "home" -> person.address.toJson
    )
}
```

JSON with Circe

Circeは、en / decode jsonのコンパイルにしたコーデックをケースクラスにします。なはのようになります。

```
import io.circe._
import io.circe.generic.auto._
import io.circe.parser._
import io.circe.syntax._

case class User(id: Long, name: String)

val user = User(1, "John Doe")

// {"id":1,"name":"John Doe"}
val json = user.asJson.noSpaces

// Right(User(1L, "John Doe"))
val res: Either[Error, User] = decode[User](json)
```

JSON with play-json

play-jsonはのをのjsonフレームワークとしてします

SBT `libraryDependencies += ""com.typesafe.play" %% "play-json" % "2.4.8"`

```
import play.api.libs.json._
import play.api.libs.functional.syntax._ // if you need DSL
```

`DefaultFormat`は、すべてのをみきするためのデフォルトがまれています。のにしてJSONをするには、フォーマットになBuilderをするか、にをいてください。

jsonをむ

```
// generates an intermediate JSON representation (abstract syntax tree)
val res = Json.parse("""{ "foo": "bar" }""") // JsValue = {"foo":"bar"}

res.as[Map[String, String]]                  // Map(foo -> bar)
res.validate[Map[String, String]]            //JsSuccess(Map(foo -> bar),)
```

jsonをく

```
val values = List("a", "b", "c")
Json.stringify(Json.toJson(values))          // ["a", "b", "c"]
```

DSL

```
val json = parse("""{ "foo": [{"foo": "bar"}]}""")
(json \ "foo").get                    //Simple path: [{"foo":"bar"}]
(json \\ "foo")                       //Recursive path:List([{"foo":"bar"}], "bar")
(json \ "foo")(0).get                  //Index lookup (for JsArrays): {"foo":"bar"}
```

に*JsSuccess* / *JsError*にするパターンマッチングを*JsSuccess*し、 *.get* 、 *array(i)*びしをけようとします。

ケースクラスのみき

```
case class Address(street: String, city: String)
case class Person(name: String, address: Address)

// create the formats and provide them implicitly
implicit val addressFormat = Json.format[Address]
implicit val personFormat = Json.format[Person]

// serialize a Person
val fred = Person("Fred", Address("Awesome Street 9", "SuperCity"))
val fredJsonString = Json.stringify(Json.toJson(Json.toJson(fred)))

val personRead = Json.parse(fredJsonString).as[Person] //Person(Fred,Address(Awesome Street
9,SuperCity))
```

のフォーマット

のなシリアルがなは、のJsonFormatをすることができますたとえば、scalaやJsonではフィールドのをなるようにするか、についてなるなをインスタンスします

```
case class Address(street: String, city: String)

// create the formats and provide them implicitly
implicit object AddressFormatCustom extends Format[Address] {
  def reads(json: JsValue): JsResult[Address] = for {
    street <- (json \ "Street").validate[String]
    city <- (json \ "City").validate[String]
  } yield Address(street, city)

  def writes(x: Address): JsValue = Json.obj(
    "Street" -> x.street,
    "City" -> x.city
  )
}
// serialize an address
val address = Address("Awesome Street 9", "SuperCity")
val addressJsonString = Json.stringify(Json.toJson(Json.toJson(address)))
//{"Street":"Awesome Street 9","City":"SuperCity"}

val addressRead = Json.parse(addressJsonString).as[Address]
//Address(Awesome Street 9,SuperCity)
```

オルタナティブ

JSONはにあなたのケースクラスのフィールドとしない`isAlive`ケースクラスに`is_alive` JSONで

```
case class User(username: String, friends: Int, enemies: Int, isAlive: Boolean)

object User {

  import play.api.libs.functional.syntax._
  import play.api.libs.json._

  implicit val userReads: Reads[User] = (
      (JsPath \ "username").read[String] and
      (JsPath \ "friends").read[Int] and
      (JsPath \ "enemies").read[Int] and
      (JsPath \ "is_alive").read[Boolean]
    ) (User.apply _)
}
```

オプションフィールドをつ**Json**

```
case class User(username: String, friends: Int, enemies: Int, isAlive: Option[Boolean])

object User {

  import play.api.libs.functional.syntax._
  import play.api.libs.json._

  implicit val userReads: Reads[User] = (
      (JsPath \ "username").read[String] and
      (JsPath \ "friends").read[Int] and
      (JsPath \ "enemies").read[Int] and
      (JsPath \ "is_alive").readNullable[Boolean]
    ) (User.apply _)
}
```

jsonのタイムスタンプをむ

UnixタイムスタンプフィールドをつJsonオブジェクトがあるとします。

```
{
  "field": "example field",
  "date": 1459014762000
}
```

```
case class JsonExampleV1(field: String, date: DateTime)
object JsonExampleV1{
  implicit val r: Reads[JsonExampleV1] = (
    (__ \ "field").read[String] and
      (__ \ "date").read[DateTime](Reads.DefaultJodaDateReads)
    )(JsonExampleV1.apply _)
}
```

カスタムケースクラスのみみ

、ののためにオブジェクトをラップすると、これをしむことができます。のjsonオブジェクトをしてください。

```
{
  "id": 91,
  "data": "Some data"
}
```

するケースクラス

```
case class MyIdentifier(id: Long)

case class JsonExampleV2(id: MyIdentifier, data: String)
```

これで、プリミティブLongをみ、あなたのidenfierにマップするだけです

```
object JsonExampleV2 {
  implicit val r: Reads[JsonExampleV2] = (
      (__ \ "id").read[Long].map(MyIdentifier) and
    (__ \ "data").read[String]
    )(JsonExampleV2.apply _)
}
```

コードhttps://github.com/pedrorijo91/scala-play-json-examples

JSONとjson4s

json4sはのをのjsonフレームワークとしてします。

SBTの

```
libraryDependencies += "org.json4s" %% "json4s-native" % "3.4.0"
//or
libraryDependencies += "org.json4s" %% "json4s-jackson" % "3.4.0"
```

```
import org.json4s.JsonDSL._
import org.json4s._
import org.json4s.native.JsonMethods._

implicit val formats = DefaultFormats
```

`DefaultFormats`は、すべてのをみきするためののがまれています。

jsonをむ

```
// generates an intermediate JSON representation (abstract syntax tree)
val res = parse("""{ "foo": "bar" }""")           // JValue = {"foo":"bar"}
res.extract[Map[String, String]]                  // Map(foo -> bar)
```

jsonをく

```
val values = List("a", "b", "c")
compact(render(values))         // ["a", "b", "c"]
```

DSL

```
json \ "foo"       //Simple path: JArray(List(JObject(List((foo,JString(bar))))))
json \\ "foo"      //Recursive path: ~List([{"foo":"bar"}], "bar")
(json \ "foo")(0)  //Index lookup (for JsArrays): JObject(List((foo,JString(bar))))
("foo" -> "bar") ~ ("field" -> "value")    // {"foo":"bar","field":"value"}
```

ケースクラスのみき

```
import org.json4s.native.Serialization.{read, write}

case class Address(street: String, city: String)
val addressString = write(Address("Awesome stree", "Super city"))
// {"street":"Awesome stree","city":"Super city"}

read[Address](addressString) // Address(Awesome stree,Super city)
//or
parse(addressString).extract[Address]
```

リストのみりときみ

またはのリストをおよびするには、のヒントをするがあります。

```
trait Location
case class Street(name: String) extends Location
case class City(name: String, zipcode: String) extends Location
case class Address(street: Street, city: City) extends Location
case class Locations (locations : List[Location])

implicit val formats = Serialization.formats(ShortTypeHints(List(classOf[Street],
classOf[City], classOf[Address])))

val locationsString = write(Locations(Street("Lavelle Street"):: City("Super city","74658")))

read[Locations](locationsString)
```

のフォーマット

```
class AddressSerializer extends CustomSerializer[Address](format => (
  {
    case JObject(JField("Street", JString(s)) :: JField("City", JString(c)) :: Nil) =>
      new Address(s, c)
  },
  {
    case x: Address => ("Street" -> x.street) ~ ("City" -> x.city)
  }
  ))

implicit val formats = DefaultFormats + new AddressSerializer
val str = write[Address](Address("Awesome Stree", "Super City"))
// {"Street":"Awesome Stree","City":"Super City"}
read[Address](str)
// Address(Awesome Stree,Super City)
```

オンラインでJSONをむ https://riptutorial.com/ja/scala/topic/2348/json

5: Quasiquotes

Examples

quasiquotesでをする

マクロに`Tree`をするには、クォーテーションをします。

```
object macro {
  def addCreationDate(): java.util.Date = macro impl.addCreationDate
}

object impl {
  def addCreationDate(c: Context)(): c.Expr[java.util.Date] = {
    import c.universe._

    val date = q"new java.util.Date()" // this is the quasiquote
    c.Expr[java.util.Date](date)
  }
}
```

それはになことができますが、しいスカラのためにされます。

オンラインでQuasiquotesをむ https://riptutorial.com/ja/scala/topic/4032/quasiquotes

6: Scala.js

き

Scala.jsは、 JavaScriptにコンパイルされたScalaポートで、はJVMでされます。なけ、コンパイルのコード、JavaScriptライブラリとのななどのがあります。

Examples

Scala.jsのconsole.log

```
println("Hello Scala.js") // In ES6: console.log("Hello Scala.js");
```

の

```
val lastNames = people.map(p => p.lastName)
// Or shorter:
val lastNames = people.map(_.lastName)
```

シンプルクラス

```
class Person(val firstName: String, val lastName: String) {
  def fullName(): String =
    s"$firstName $lastName"
}
```

コレクション

```
val personMap = Map(
  10 -> new Person("Roger", "Moore"),
  20 -> new Person("James", "Bond")
)
val names = for {
  (key, person) <- personMap
  if key > 15
} yield s"$key = ${person.firstName}"
```

DOMの

```
import org.scalajs.dom
import dom.document

def appendP(target: dom.Node, text: String) = {
  val pNode = document.createElement("p")
  val textNode = document.createTextNode(text)
  pNode.appendChild(textNode)
```

```
  target.appendChild(pNode)
}
```

SBTでする

Sbt

```
libraryDependencies += "org.scala-js" %%% "scalajs-dom" % "0.9.1" // (Triple %%%)
```

ランニング

```
sbt run
```

コンパイルをしてする

```
sbt ~run
```

のJavaScriptファイルにコンパイルする

```
sbt fastOptJS
```

オンラインでScala.jsをむ https://riptutorial.com/ja/scala/topic/9426/scala-js

7: ScalaCheckによるテスト

き

ScalaCheckは、Scalaでかれたライブラリであり、ScalaやJavaプログラムのプロパティベースのテストにされます。 ScalaCheckはもともとHaskellライブラリのQuickCheckにされていましたが、のでもテストしました。

ScalaCheckには、Scalaランタイムのはなく、Scalaビルドツールであるsbtでうまくします。また、ScalaTestとspecs2のテストフレームワークににされています。

Examples

スケーラビリティとエラーメッセージをむScalacheck

Scatatestをするscalacheckのに4つのテストがあります。

- "show pass example" - それがする
- "カスタムエラーメッセージがされないなをする" - がなくメッセージにしました。 `&&`ブールがされています
- "にエラーメッセージがされたをする" - `"argument" |:` にエラーメッセージがされる`&&`わりにProps.allメソッドがされる
- "コマンドのエラーメッセージをする" - コマンドのエラーメッセージ `"command" |: &&`わりにProps.allメソッドがされる

```
import org.scalatest.prop.Checkers
import org.scalatest.{Matchers, WordSpecLike}

import org.scalacheck.Gen._
import org.scalacheck.Prop._
import org.scalacheck.Prop

object Splitter {
  def splitLineByColon(message: String): (String, String) = {
    val (command, argument) = message.indexOf(":") match {
      case -1 =>
        (message, "")
      case x: Int =>
        (message.substring(0, x), message.substring(x + 1))
    }
    (command.trim, argument.trim)
  }

  def splitLineByColonWithBugOnCommand(message: String): (String, String) = {
    val (command, argument) = splitLineByColon(message)
    (command.trim + 2, argument.trim)
  }

  def splitLineByColonWithBugOnArgument(message: String): (String, String) = {
```

```
    val (command, argument) = splitLineByColon(message)
    (command.trim, argument.trim + 2)
  }
}

class ScalaCheckSpec extends WordSpecLike with Matchers with Checkers {

  private val COMMAND_LENGTH = 4

  "ScalaCheckSpec " should {
```

```
    "show pass example" in {
      check {
        Prop.forAll(listOfN(COMMAND_LENGTH, alphaChar), alphaStr) {
          (chars, expArgument) =>
            val expCommand = new String(chars.toArray)
            val line = s"$expCommand:$expArgument"
            val (c, p) = Splitter.splitLineByColon(line)
            Prop.all("command" |: c =? expCommand, "argument" |: expArgument =? p)
        }

      }
    }
```

```
"show simple example without custom error message " in {
  check {
    Prop.forAll(listOfN(COMMAND_LENGTH, alphaChar), alphaStr) {
      (chars, expArgument) =>
        val expCommand = new String(chars.toArray)
        val line = s"$expCommand:$expArgument"
        val (c, p) = Splitter.splitLineByColonWithBugOnArgument(line)
        c === expCommand && expArgument === p
    }

  }
}
```

```
"show example with error messages on argument" in {
  check {
    Prop.forAll(listOfN(COMMAND_LENGTH, alphaChar), alphaStr) {
      (chars, expArgument) =>
        val expCommand = new String(chars.toArray)
        val line = s"$expCommand:$expArgument"
        val (c, p) = Splitter.splitLineByColonWithBugOnArgument(line)
        Prop.all("command" |: c =? expCommand, "argument" |: expArgument =? p)
    }

  }
}
```

```
"show example with error messages on command" in {
  check {
    Prop.forAll(listOfN(COMMAND_LENGTH, alphaChar), alphaStr) {
      (chars, expArgument) =>
        val expCommand = new String(chars.toArray)
        val line = s"$expCommand:$expArgument"
        val (c, p) = Splitter.splitLineByColonWithBugOnCommand(line)
        Prop.all("command" |: c =? expCommand, "argument" |: expArgument =? p)
```

```
    }

  }
}
```

フラグメント

```
[info] - should show example // passed
[info] - should show simple example without custom error message  *** FAILED ***
[info]    (ScalaCheckSpec.scala:73)
[info]     Falsified after 0 successful property evaluations.
[info]     Location: (ScalaCheckSpec.scala:73)
[info]     Occurred when passed generated values (
[info]       arg0 = List(), // 3 shrinks
[info]       arg1 = ""
[info]     )
[info] - should show example with error messages on argument *** FAILED ***
[info]    (ScalaCheckSpec.scala:86)
[info]     Falsified after 0 successful property evaluations.
[info]     Location: (ScalaCheckSpec.scala:86)
[info]     Occurred when passed generated values (
[info]       arg0 = List(), // 3 shrinks
[info]       arg1 = ""
[info]     )
[info]     Labels of failing property:
[info]       Expected "" but got "2"
[info]       argument
[info] - should show example with error messages on command *** FAILED ***
[info]    (ScalaCheckSpec.scala:99)
[info]     Falsified after 0 successful property evaluations.
[info]     Location: (ScalaCheckSpec.scala:99)
[info]     Occurred when passed generated values (
[info]       arg0 = List(), // 3 shrinks
[info]       arg1 = ""
[info]     )
[info]     Labels of failing property:
[info]       Expected "2" but got ""
[info]       command
```

オンラインでScalaCheckによるテストをむ https://riptutorial.com/ja/scala/topic/9430/scalacheck によるテスト

8: ScalaTestによるテスト

Examples

Hello World Specテスト

`HelloWorldSpec.scala`というのファイルで、 `src/test/scala`ディレクトリにテストクラスをします。これをファイルのにれます

```
import org.scalatest.{FlatSpec, Matchers}

class HelloWorldSpec extends FlatSpec with Matchers {

  "Hello World" should "not be an empty String" in {
      val helloWorld = "Hello World"
      helloWorld should not be ("")
  }
}
```

- このでは、 ScalaTestライブラリのである`FlatSpec`と`Matchers`をしています。
- `FlatSpec`は、テストを**Behavior-Driven DevelopmentBDD**スタイルですることをにします。このスタイルでは、のコードのされるをするためにセンテンスがされます。このテストは、コードがそのにっていることをします。 については、ドキュメントをしてください 。

テストチートシート

セットアップ

のテストでは、これらのをとしてしています。

```
val helloWorld = "Hello World"
val helloWorldCount = 1
val helloWorldList = List("Hello World", "Bonjour Le Monde")
def sayHello = throw new IllegalStateException("Hello World Exception")
```

チェック

された`val`をするには

```
helloWorld shouldBe a [String]
```

ここでは`String`をするためにされることにしてください。

チェック

をテストするには

```
helloWorld shouldEqual "Hello World"
helloWorld should === ("Hello World")
helloWorldCount shouldEqual 1
helloWorldCount shouldBe 1
helloWorldList shouldEqual List("Hello World", "Bonjour Le Monde")
helloWorldList === List("Hello World", "Bonjour Le Monde")
```

しくない

をテストするには

```
helloWorld should not equal "Hello"
helloWorld !== "Hello"
helloWorldCount should not be 5
helloWorldList should not equal List("Hello World")
helloWorldList !== List("Hello World")
helloWorldList should not be empty
```

さチェック

さおよび/またはサイズをするには

```
helloWorld should have length 11
helloWorldList should have size 2
```

チェック

のタイプとメッセージをするには

```
val exception = the [java.lang.IllegalStateException] thrownBy {
  sayHello
}
exception.getClass shouldEqual classOf[java.lang.IllegalStateException]
exception.getMessage should include ("Hello World")
```

SBTにScalaTestライブラリをめる

SBTをしてライブラリのをするは、`build.sbt`にこれをし`build.sbt`。

```
libraryDependencies += "org.scalactic" %% "scalactic" % "3.0.0"
libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.0" % "test"
```

は、ScalaTestサイトをしてください。

オンラインでScalaTestによるテストをむ https://riptutorial.com/ja/scala/topic/5506/scalatestによるテスト

9: Scalaの

Examples

みみ

Scalaにはのビルトインがありますみのルールをつmethods / language。

タイプ	シンボル	
	+ - * / %	a + b
	== != > < >= <=	a > b
	&& & || | !	a && b
ビットワイズ	& | ^ ~ << >> >>>	a & b、 ~a、 a >>> b
	= += -= *= /= %= <<= >>= &= ^= = =	a += b

ScalaはJavaとじをちます

のメソッドは:およびにバインドします。したがって、 `value :: list` . `value :: list` `list.::(value)` したびしは、をして `value :: list` としてできます。 `1 :: 2 :: 3 :: Nil` は `1 :: (2 :: (3 :: Nil))` とじです

のオーバーロード

Scalaでは、あなたのをすることができます

```
class Team {
   def +(member: Person) = ...
}
```

のをすると、のようにできます。

```
ITTeam + Jack
```

または

```
ITTeam.+(Jack)
```

をするには、そのに `unary_` けることができます。たとえば `unary_!`

```
class MyBigInt {
```

```
   def unary_! = ...
}

var a: MyBigInt = new MyBigInt
var b = !a
```

オペレータの

カテゴリー	オペレーター	
Postfix	() []	からへ
	! ~	からへ
	* / %	からへ
	+ -	からへ
シフト	>> >>> <<	からへ
リレーショナル	> >= < <=	からへ
	== !=	からへ
ビットおよび	&	からへ
ビットのxor	^	からへ
ビットまたは	|	からへ
および	&&	からへ
または	||	からへ
りて	= += -= *= /= %= >>= <<= &= ^= = =	からへ
コンマ	,	からへ

Scalaでプログラミングすると、ののについてのようなアウトラインがられます。 Eg >は>>>のです

オペレーター
のすべての
* / %
+ -

オペレーター
:
= !
< >
&
^
|
すべての
すべての

が=でわり、<= 、 >= 、 ==または!=いずれかでない、このルールのは += 、 *=のはなりてとじです。すれば、のオペレータのものよりい。

オンラインでScalaのをむ https://riptutorial.com/ja/scala/topic/6604/scalaの

10: Scalaの

Examples

Linuxでdpkgで

UbuntuをむDebianベースのディストリビューションでは、もなは`.deb`インストールファイルをうことです。 ScalaのWebサイトにアクセスしてください 。インストールするバージョンをし、スクロールして`scala-xxxdeb`ます。

コマンドラインからscala debをインストールすることができます

```
sudo dpkg -i scala-x.x.x.deb
```

ターミナルコマンドプロンプトにしくインストールされていることをするには、のようにします。

```
which scala
```

されるは、PATHにしたものとでなければなりません。スカラーがしていることをするには

```
scala
```

これによりScalaのREPLがし、バージョンをしますダウンロードしたバージョンとするはずです。

Ubuntuのダウンロードとによるインストール

`curl` Lightbendからあなたのみのバージョンをダウンロードしてください

```
curl -O http://downloads.lightbend.com/scala/2.xx.x/scala-2.xx.x.tgz
```

`tar`ファイルを`/usr/local/share`または`/opt/bin`します。

```
unzip scala-2.xx.x.tgz
mv scala-2.xx.x /usr/local/share/scala
```

`~/.profile`または`~/.bash_profile`または`~/.bashrc`にこれらのファイルの1つにこのテキストをめて`PATH`をしてください

```
$SCALA_HOME=/usr/local/share/scala
export PATH=$SCALA_HOME/bin:$PATH
```

ターミナルコマンドプロンプトにしくインストールされていることをするには、のようにします

。

```
which scala
```

されるは、 `PATH`にしたものとでなければなりません。 `scala`がしていることをするには

```
scala
```

これによりScalaのREPLがし、バージョンをしますダウンロードしたバージョンとするはずです。

Mac OSXでMacports

MacPortsがインストールされたMac OSXコンピュータでは、ターミナルウィンドウをいてのようにします。

```
port list | grep scala
```

なすべてのScalaパッケージがされます。インストールするにはこのではScalaの2.11バージョン

```
sudo port install scala2.11
```

のバージョンをインストールするは、 `2.11`がされることがあります。

すべてのがにインストールされ、 `$PATH`パラメータがされます。すべてのをするには

```
which scala
```

これにより、Scalaインストールへのパスがされます。

```
scala
```

これにより、ScalaのREPLがき、インストールされているバージョンがされます。

オンラインでScalaのをむ https://riptutorial.com/ja/scala/topic/2921/scalaの

11: Whileループ

- whileboolean_expression{block_expression}
- do {block_expression} whileboolean_expressionwhile

パラ メーター

パラ メータ	
`boolean_expression`	`true`または`false`とされる。
`block_expression`	`boolean_expression`が`true`とされたにされるまたはのセット。

`do-while`ループと`do-while`ループのないは`while`ループするがあるかどうかをするに`block_expression`をするかどうかです。

なぜなら`while`と`do-while`ループにするにる`false`するために、らはしばしばループのでされ、ループでするをとしています。

Examples

Whileループ

```
var line = 0
var maximum_lines = 5

while (line < maximum_lines) {
  line = line + 1
  println("Line number: " + line)
}
```

Do-Whileループ

```
var line = 0
var maximum_lines = 5

do {
  line = line + 1
  println("Line number: " + line)
} while (line < maximum_lines)
```

`do` / `while`ループは、プログラミングではまれにしかされませんが、のでられるように、 `break` / `continue`のサポートのをするためにできます。

```
if(initial_condition) do if(filter) {
  ...
} while(continuation_condition)
```

オンラインでWhileループをむ https://riptutorial.com/ja/scala/topic/650/whileループ

12: XML

Examples

BeautifyまたはPretty-Print XML

`PrettyPrinter`ユーティリティは、XMLドキュメントを「きれいに」します。のコードスニペットは、フォーマットされていないxmlをきれいにします。

```
import scala.xml.{PrettyPrinter, XML}
val xml = XML.loadString("<a>Alana<b><c>Beth</c><d>Catie</d></b></a>")
val formatted = new PrettyPrinter(150, 4).format(xml)
print(formatted)
```

これにより、ページ`150`、インデント`4`のをしてコンテンツがされます。

```
<a>
    Alana
    <b>
        <c>Beth</c>
        <d>Catie</d>
    </b>
</a>
```

`XML.loadFile("nameoffile.xml")`をすると、ではなくファイルからxmlをロードできます。

オンラインでXMLをむ https://riptutorial.com/ja/scala/topic/1453/xml

13: インプリシット

- のval xT = ???

のクラスでは、のにカスタムメソッドをすることができ、コードをすることなくコードをすることなくをさせることができます。

のをしてのクラスをさせることは、しばしば「ライブラリをさせる」パターンとばれます。

なクラスの

1. のクラスは、のクラス、オブジェクト、またはにのみすることがあります。
2. のクラスは、でない1コンストラクタ·パラメータを1つだけつことができます。
3. のクラスとじをつじスコープにのオブジェクト、クラス、、またはクラスメンバーがしないことがあります。

Examples

の

なにより、あるタイプのオブジェクトをのタイプににすることができます。これにより、コードはオブジェクトをののオブジェクトとしてうことができます。

```
case class Foo(i: Int)

// without the implicit
Foo(40) + 2    // compilation-error (type mismatch)

// defines how to turn a Foo into an Int
implicit def fooToInt(foo: Foo): Int = foo.i

// now the Foo is converted to Int automatically when needed
Foo(40) + 2    // 42
```

はですこの、`42`を`Foo(42)`することはできません。これをうには、2ののをするがあります。

```
implicit def intToFoo(i: Int): Foo = Foo(i)
```

これは、floatをなどにできるメカニズムであることにしてください。

> のコンバージョンは、がこっているのかをするため、えめにするがあります。のをすることによるのがないり、メソッドびしによるなをすることがベストプラクティスです。

なにはパフォーマンスにきなはありません。

Scalaはに、JavaからScalaへのすべてのをむ、`scala.Predef`さまざまなのを`scala.Predef`します。

これらは、デフォルトではファイルのコンパイルにまれます。

のパラメータ

のパラメータは、のパラメータをスコープでし、そののをするすべてのにするがあるにです。

のびしは、のようになります。

```
// import the duration methods
import scala.concurrent.duration._

// a normal method:
def doLongRunningTask(timeout: FiniteDuration): Long = timeout.toMillis

val timeout = 1.second
// timeout: scala.concurrent.duration.FiniteDuration = 1 second

// to call it
doLongRunningTask(timeout) // 1000
```

ここでは、すべてのタイムアウトがあるいくつかのメソッドがあり、じタイムアウトをしてこれらのメソッドをすべてびすとします。のとしてタイムアウトをできます。

```
// import the duration methods
import scala.concurrent.duration._

// dummy methods that use the implicit parameter
def doLongRunningTaskA()(implicit timeout: FiniteDuration): Long = timeout.toMillis
def doLongRunningTaskB()(implicit timeout: FiniteDuration): Long = timeout.toMillis

// we define the value timeout as implicit
implicit val timeout: FiniteDuration = 1.second

// we can now call the functions without passing the timeout parameter
doLongRunningTaskA() // 1000
doLongRunningTaskB() // 1000
```

これがするは、scalacコンパイラがにマークされスコープのをし、そのがのパラメータのとすることです。それがつかったは、それをのパラメーターとしてします。

スコープでじタイプの2つののをすると、これはしません。

エラーメッセージをカスタマイズするには、に`implicitNotFound`アノテーションをします。

```
@annotation.implicitNotFound(msg = "Select the proper implicit value for type M[${A}]!")
case class M[A](v: A) {}

def usage[O](implicit x: M[O]): O = x.v

//Does not work because no implicit value is present for type `M[Int]`
//usage[Int]   //Select the proper implicit value for type M[Int]!
implicit val first: M[Int] = M(1)
usage[Int]     //Works when `second` is not in scope
implicit val second: M[Int] = M(2)
```

```
//Does not work because more than one implicit values are present for the type `M[Int]`
//usage[Int]   //Select the proper implicit value for type M[Int]!
```

タイムアウトはのユースケースです。たとえばAkkaの 、ActorSystemはにほとんどのはじであるため、はにされます。のユースケースは、のようなクラスにしているFPライブラリでもに、ライブラリのになりscalaz 、 や 。

> *Int* 、 *Long* 、 *String*などのでなパラメータをすることは、にいとみなされます。をき 、コードをみにくくするからです。

のクラス

のクラスは、にされたクラスにしいメソッドをすることをにします。

`String`クラスには`withoutVowels`メソッドはありません。これはのようにできます

```
object StringUtil {
  implicit class StringEnhancer(str: String) {
    def withoutVowels: String = str.replaceAll("[aeiou]", "")
  }
}
```

のクラスには、したい `String` をつのコンストラクタパラメータ `str` があり、 `withoutVowels` に ""するメソッドがまれています。しくされたメソッドは、エンハンスドタイプエンハンスタイプがスコープのでできるようになりました。

```
import StringUtil.StringEnhancer // Brings StringEnhancer into implicit scope

println("Hello world".withoutVowels) // Hll wrld
```

では、のクラスは、のように、されたからのクラスへののをします。

```
implicit def toStringEnhancer(str: String): StringEnhancer = new StringEnhancer(str)
```

のクラスは、ランタイムオブジェクトのをけ、ランタイムオーバーヘッドをするために、 Value クラスとしてされることがよくあります。

```
implicit class StringEnhancer(val str: String) extends AnyVal {
    /* conversions code here */
}
```

のされたでは、 `StringEnhancer`しいインスタンスは、 `withoutVowels`メソッドがびされるたびにするはありません。

'に'のパラメータをしてする

ののパラメータをつのパラメータリストをします。

```
case class Example(p1:String, p2:String)(implicit ctx1:SomeCtx1, ctx2:SomeCtx2)
```

ここで、のインスタンスの1つがでない `SomeCtx1` 、ののインスタンスがすべてスコープにあると`SomeCtx1`すると、クラスのインスタンスをするには`SomeCtx1`インスタンスをするがあります。

これは`implicitly`キーワードをしてスコープののインスタンスをにしながらうことができます

```
Example("something","somethingElse")(new SomeCtx1(), implicitly[SomeCtx2])
```

REPLのインプリシット

REPLセッションにスコープの`implicits`をすべてするには、のようにします。

```
scala> :implicits
```

`Predef.scala`されたなもめるには

```
scala> :implicits -v
```

があり、それにされるすべてのきえルールのをしたいはをむ

```
scala> reflect.runtime.universe.reify(expr) // No quotes. reify is a macro operating directly
on code.
```

```
scala> import reflect.runtime.universe._
scala> reify(Array("Alice", "Bob", "Eve").mkString(", "))
resX: Expr[String] = Expr[String](Predef.refArrayOps(Array.apply("Alice", "Bob",
"Eve")(Predef.implicitly)).mkString(", "))
```

オンラインでインプリシットをむ https://riptutorial.com/ja/scala/topic/1732/インプリシット

14: ヴァール、ヴァル、デフ

valはにであるため、コードにされているはいつでも「インプレース」でされます。これは、なクラスおよびでされる、くべきましくないをきこすがあります。

えば、ラップされたIntをするPlusOneというをりたいとしましょう。 Intはであるため、に1をえたがし、それはされないので、にはvalです。ただし、このようにすると、しないがします。

```
trait PlusOne {
    val i:Int

    val incr = i + 1
}

class IntWrapper(val i: Int) extends PlusOne
```

どんなにどのようなiあなたがIntWrapperびし、と.incrに1をします。されたオブジェクトにこのヴァルためであるincrクラスのに、 にされ、そのでiのデフォルトをっています0。 のでは、 Nil 、 nullなどのがされているがあります。

なは、フィールドにするのにしてvalをすることをけることです。わりに、になるまでされないlazy valや、びされるたびにされるdefをします。ことにしてくださいlazy valですることをなくされたvalがするに、じエラーがします。

Scala-Jsでかれているが、じがされるバイディングがここにあります。

Examples

ヴァール、ヴァル、デフ

var

varはで、Javaなどののにています。されたオブジェクトがvarがされたとじであるり、さまざまなオブジェクトをvarににりてることができます。

```
scala> var x = 1
x: Int = 1

scala> x = 2
x: Int = 2

scala> x = "foo bar"
<console>:12: error: type mismatch;
 found   : String("foo bar")
 required: Int
       x = "foo bar"
       ^
```

のでは、 `var`のは、ののりてがえられたときにコンパイラによってされたことにしてください。

ヴァル

`val`はです。したがって、すでにりてられている`val`しいオブジェクトをりてることはできません。

```
scala> val y = 1
y: Int = 1

scala> y = 2
<console>:12: error: reassignment to val
       y = 2
         ^
```

しかし、 `val`すオブジェクトはではありません。そのオブジェクトはされるかもしれません

```
scala> val arr = new Array[Int](2)
arr: Array[Int] = Array(0, 0)

scala> arr(0) = 1

scala> arr
res1: Array[Int] = Array(1, 0)
```

def

`def`はメソッドをします。メソッドをりてすることはできません。

```
scala> def z = 1
z: Int

scala> z = 2
<console>:12: error: value z_= is not a member of object $iw
       z = 2
       ^
```

のでは、 `val y`と`def z`はじをします。しかし、 `def`はびされたときにされますが、 `val`または`var`はりてにされます。にがあると、がなることがあります。

```
scala> val a = {println("Hi"); 1}
Hi
a: Int = 1

scala> def b = {println("Hi"); 1}
b: Int

scala> a + 1
res2: Int = 2
```

```
scala> b + 1
Hi
res3: Int = 2
```

はであるため、 `val` / `var` / `def`することができます。それはとじでします

```
scala> val x = (x: Int) => s"value=$x"
x: Int => String = <function1>

scala> var y = (x: Int) => s"value=$x"
y: Int => String = <function1>

scala> def z = (x: Int) => s"value=$x"
z: Int => String

scala> x(1)
res0: String = value=1

scala> y(2)
res1: String = value=2

scala> z(3)
res2: String = value=3
```

レ イジ―ヴァル

`lazy val`である`val` 、それがにアクセスされるまでされます。そのの、それはの`val`ようにします。

これをするには、 `val`に`lazy`キ―ワ―ドをします。たとえば、REPLをして

```
scala> lazy val foo = {
     |   println("Initializing")
     |   "my foo value"
     | }
foo: String = <lazy>

scala> val bar = {
     |   println("Initializing bar")
     |   "my bar value"
     | }
Initializing bar
bar: String = my bar value

scala> foo
Initializing
res3: String = my foo value

scala> bar
res4: String = my bar value

scala> foo
res5: String = my foo value
```

このでは、をします。 `lazy val`がされると、まだされていないびしが`foo`にされます。の`val`がされ

ると、 `println`びしがされ、が`bar`りてられ`bar`。 はじめて`foo`すると、 `println`されるのがえますが、2にされたときはされません。に、 `bar`がされると、 `println` executeがされているときにのみされます。

'け'をうべきとき

1. はコストがく、 **`val`**はまれです。

```
lazy val tiresomeValue = {(1 to 1000000).filter(x => x % 113 == 0).sum}
if (scala.util.Random.nextInt > 1000) {
  println(tiresomeValue)
}
```

`tiresomeValue`はにがかかり、にされるわけではありません。それを`lazy val`にすると、ながされます。

2. の

インスタンスににするがある2つのオブジェクトをつをてみましょう。

```
object comicBook {
  def main(args:Array[String]): Unit = {
    gotham.hero.talk()
    gotham.villain.talk()
  }
}

class Superhero(val name: String) {
  lazy val toLockUp = gotham.villain
  def talk(): Unit = {
    println(s"I won't let you win ${toLockUp.name}!")
  }
}

class Supervillain(val name: String) {
  lazy val toKill = gotham.hero
  def talk(): Unit = {
    println(s"Let me loosen up Gotham a little bit ${toKill.name}!")
  }
}

object gotham {
  val hero: Superhero = new Superhero("Batman")
  val villain: Supervillain = new Supervillain("Joker")
}
```

キーワード`lazy`なければ、それぞれのオブジェクトをオブジェクトのメンバーにすることはできません。そのようなプログラムをすると、 `java.lang.NullPointerException`がし`java.lang.NullPointerException`。 `lazy`をすることで、にをりてることができます。されていないをつことをするはありません。

オーバーロードDef

がなるは、 `def`をオーバーロードすることができます

```
def printValue(x: Int) {
  println(s"My value is an integer equal to $x")
}

def printValue(x: String) {
  println(s"My value is a string equal to '$x'")
}

printValue(1)  // prints "My value is an integer equal to 1"
printValue("1") // prints "My value is a string equal to '1'"
```

これは、クラス、、オブジェクトであろうとなかろうと、じようにします。

きパラメータ

`def`をびすときは、パラメータをにでりてることができます。そうすることは、らがしくされるがないことをする。たとえば、 `printUs()`をのようにします。

```
// print out the three arguments in order.
def printUs(one: String, two: String, three: String) =
   println(s"$one, $two, $three")
```

では、これらのでのもののでびすことができます

```
printUs("one", "two", "three")
printUs(one="one", two="two", three="three")
printUs("one", two="two", three="three")
printUs(three="three", one="one", two="two")
```

これは、その`one`, `two`, `three`すべてのにされています。

すべてのにがいていない、のはでします。されたのろには、のいていないはありません。

```
printUs("one", two="two", three="three") // prints 'one, two, three'
printUs(two="two", three="three", "one") // fails to compile: 'positional after named
argument'
```

オンラインでヴァール、ヴァル、デフをむ https://riptutorial.com/ja/scala/topic/3155/ヴァール-ヴァル-デフ

15: エラー

Examples

す

`map` 、 `getOrElse` 、 および`flatMap`で`getOrElse`て`flatMap`

```
import scala.util.Try

val i = Try("123".toInt)     // Success(123)
i.map(_ + 1).getOrElse(321)  // 124

val j = Try("abc".toInt)     // Failure(java.lang.NumberFormatException)
j.map(_ + 1).getOrElse(321)  // 321

Try("123".toInt) flatMap { i =>
  Try("234".toInt)
    .map(_ + i)
}                            // Success(357)
```

パターンマッチングで`Try`をう

```
Try(parsePerson("John Doe")) match {
    case Success(person) => println(person.surname)
    case Failure(ex) => // Handle error ...
}
```

いずれか

エラー/のためのなるデータ

```
def getPersonFromWebService(url: String): Either[String, Person] = {

    val response = webServiceClient.get(url)

    response.webService.status match {
        case 200 => {
            val person = parsePerson(response)
            if(!isValid(person)) Left("Validation failed")
            else Right(person)
        }

        case _ => Left(s"Request failed with error code $response.status")
    }
}
```

いずれかのでのパターンマッチング

```
getPersonFromWebService("http://some-webservice.com/person") match {
```

```
    case Left(errorMessage) => println(errorMessage)
    case Right(person) => println(person.surname)
}
```

いずれかのをOptionにする

```
val maybePerson: Option[Person] = getPersonFromWebService("http://some-
webservice.com/person").right.toOption
```

オプション

`null`をくし、のJavaコードとのがないり、されない`null`。そのわりに、のがか `Some` またはもない `None` のいずれかであるに、 `Option`をするがあります。

try-catchブロックはエラーにしていますが、がにもさない、 `Option`はいやすく、シンプルです。

`Option[T]`は`Some(value)` `T`のをむまたは`None`

```
def findPerson(name: String): Option[Person]
```

がつからないは、 `None`をすことができます。それのは、 `Person`オブジェクトをむ`Some`のオブジェクトがされます。にすのは、 `Option`タイプのオブジェクトをするです。

パターンマッチング

```
findPerson(personName) match {
    case Some(person) => println(person.surname)
    case None => println(s"No person found with name $personName")
}
```

mapとgetOrElseをう

```
val name = findPerson(personName).map(_.firstName).getOrElse("Unknown")
println(name) // Prints either the name of the found person or "Unknown"
```

フォールドをう

```
val name = findPerson(personName).fold("Unknown")(_.firstName)
// equivalent to the map getOrElse example above.
```

Javaへの

のために`Option`タイプをヌルなJavaタイプにするがある

```
val s: Option[String] = Option("hello")
s.orNull           // "hello": String
s.getOrElse(null)  // "hello": String

val n: Option[Int] = Option(42)
n.orNull           // compilation failure (Cannot prove that Null <:< Int.)
n.getOrElse(null)  // 42
```

のエラー

Futureからexceptionがスローされると、それをするrecoverをうことができます。

えば、

```
def runFuture: Future = Future { throw new FairlyStupidException }

val itWillBeAwesome: Future = runFuture
```

... Futureから Exceptionをげる。しかし、 FairlyStupidExceptionのExceptionがいでることをすることができるので、このケースをにエレガントなでできます。

```
val itWillBeAwesomeOrIllRecover = runFuture recover {
    case stupid: FairlyStupidException =>
         BadRequest("Another stupid exception!")
}
```

ごのように、 recoverメソッドはすべてのThrowableのドメインのPartialFunctionであるため、ののだけをし、りのをFutureスタックのレベルののエテルにすることができます。

これは、 Futureコンテキストでのコードをするのとであることにしてください。

```
def runNotFuture: Unit = throw new FairlyStupidException

try {
    runNotFuture
} catch {
    case e: FairlyStupidException => BadRequest("Another stupid exception!")
}
```

Futureのでされたをすることは、らがもっとがあるのでです。らはあなたのにすべてをるわけではありません。なぜなら、それらはなるコンテキストとスレッドでされるためです。したがって、ログやログにかかないには、。

try-catchの

のようななにえてTry 、 Optionおよび Eitherエラーのため、Scalaはまた、には、にブロックのあるのtry-catchをして、Javaのにたをサポートしています。 catchはパターンマッチです

```
try {
  // ... might throw exception
```

```
} catch {
  case ioe: IOException => ... // more specific cases first
  case e: Exception => ...
  // uncaught types will be thrown
} finally {
  // ...
}
```

をいずれかのまたはオプションにする

を`Either`または`Option`にするには、 `scala.util.control.Exception`でされるメソッドをできます。

```
import scala.util.control.Exception._

val plain = "71a"
val optionInt: Option[Int] = catching(classOf[java.lang.NumberFormatException]) opt {
plain.toInt }
val eitherInt = Either[Throwable, Int] = catching(classOf[java.lang.NumberFormatException])
either { plain.toInt }
```

オンラインでエラーをむ https://riptutorial.com/ja/scala/topic/910/エラー

16: オプションクラス

- class Some [+ T]valueTはOption [T]をします。
- オブジェクトなしオプション[もありません]
- オプション[T]T

  されたにじて`Some(value)`または`None`をするコンストラクタ。

Examples

コレクションとしてのオプション

`Option` Sがに0または1のアイテムをコレクションとしてオプションをすることによってすることができるいくつかのなのをっている-ここで`None`のコレクションのようにせず、 `Some(x)` 、のアイテム、のコレクションのようにるう`x`。

```
val option: Option[String] = ???

option.map(_.trim) // None if option is None, Some(s.trim) if Some(s)
option.foreach(println) // prints the string if it exists, does nothing otherwise
option.forall(_.length > 4) // true if None or if Some(s) and s.length > 4
option.exists(_.length > 4) // true if Some(s) and s.length > 4
option.toList // returns an actual list
```

Nullのわりに Optionをう

Javaおよびのでは、 `null`をすることは、にがされていないことをすなです。 Scalaでは、 `Option`をするが`null`をするよりもされ`null`。 `Option` は 、 `null` のあるをラップし`null`。

`None`はnullをラップする`Option`サブクラスです。 `Some`は、nullのをラップする`Option`サブクラスです。

のりしはです。

```
val nothing = Option(null) // None
val something = Option("Aren't options cool?") // Some("Aren't options cool?")
```

これは、nullをすのあるJavaライブラリをびすときのなコードです。

```
val resource = Option(JavaLib.getResource())
// if null, then resource = None
// else resource = Some(resource)
```

`getResource()`が`null`をした、 `resource`は`None`オブジェクトになります。それのは、 `Some(resource)`

オブジェクトになります。 `Option`をするためのましいは、 `Option`タイプでなをすることです。たとえば、が`None`でないかどうかをチェックする `value == null`と、 `isDefined`をします。

```
val resource: Option[Resource] = Option(JavaLib.getResource())
if (resource.isDefined) {  // resource is `Some(_)` type
  val r: Resource = resource.get
  r.connect()
}
```

に、 `null`をチェックするには、のようにし`null`。

```
val resource: Option[Resource] = Option(JavaLib.getResource())
if (resource.isEmpty) { // resource is `None` type.
  System.out.println("Resource is empty! Cannot connect.")
}
```

`Option`をモナドとしてい、 `foreach`をって`Option`ラップされたにするきをうことがましい'な' `Option.get`メソッドをせずに。

```
val resource: Option[Resource] = Option(JavaLib.getResource())
resource foreach (r => r.connect())
// if r is defined, then r.connect() is run
// if r is empty, then it does nothing
```

`Resource`インスタンスがな `Option[Resource]`インスタンスにして、 `Option`をしてnullからすることはできます。ここで`getOrElse`メソッドはデフォルトをします

```
lazy val defaultResource = new Resource()
val resource: Resource = Option(JavaLib.getResource()).getOrElse(defaultResource)
```

JavaコードはScalaの`Option`にうことができないので、をJavaコードにすときは、 `Option`をアンラップして、なには`null`またはなデフォルトをしてください。

```
val resource: Option[Resource] = ???
JavaLib.sendResource(resource.orNull)
JavaLib.sendResource(resource.getOrElse(defaultResource)) //
```

`Option`は、のをむか、まったくをたないデ―タです。 `Option`は、0または1のコレクションとえることができます。

Optionは2つのをつクラスです `Some`と`None`。

`Some`はのをみ、 `None`はをみません。

`Option`は、なのをすために`null`をするでです。これは、 `NullPointerException` `FlatMap` 、 `Map` 、 `FlatMap`などのコンビネ―タをしてをさないくののをにします。

マップの

```
val countries = Map(
  "USA" -> "Washington",
  "UK" -> "London",
  "Germany" -> "Berlin",
  "Netherlands" -> "Amsterdam",
  "Japan" -> "Tokyo"
)

println(countries.get("USA")) // Some(Washington)
println(countries.get("France")) // None
println(countries.get("USA").get) // Washington
println(countries.get("France").get) // Error: NoSuchElementException
println(countries.get("USA").getOrElse("Nope")) // Washington
println(countries.get("France").getOrElse("Nope")) // Nope
```

`Option[A]`はされているためできません。したがって、セマンティクスはしており、することができます。

のためのオプション

`Option`は、 `flatMap`メソッドがあります。これは、それらがのためにできることをします。このようにして、することなく、 `Option`にりむためにのをびすことができます。

```
val firstOption: Option[Int] = Option(1)
val secondOption: Option[Int] = Option(2)

val myResult = for {
  firstValue <- firstOption
  secondValue <- secondOption
} yield firstValue + secondValue
// myResult: Option[Int] = Some(3)
```

の1つが`None`の、のは`None`になります。

```
val firstOption: Option[Int] = Option(1)
val secondOption: Option[Int] = None

val myResult = for {
  firstValue <- firstOption
  secondValue <- secondOption
} yield firstValue + secondValue
// myResult: Option[Int] = None
```

このパターンは`Monad`とばれるのためによりにされています。 よりくのは、と`Monad`にするペ―ジでできるはずです

に、なるモナドをしてすることはです。しかし、 `Option`はに`Iterable`にできるので、 `.toIterable`メソッドをびすことで、 `Option`と`Iterable`にみわせることができます。

```
val option: Option[Int] = Option(1)
val iterable: Iterable[Int] = Iterable(2, 3, 4, 5)

// does NOT compile since we cannot mix Monads in a for comprehension
// val myResult = for {
//   optionValue <- option
//   iterableValue <- iterable
//} yield optionValue + iterableValue

// It does compile when adding a .toIterable on the option
val myResult = for {
  optionValue <- option.toIterable
  iterableValue <- iterable
} yield optionValue + iterableValue
// myResult: Iterable[Int] = List(2, 3, 4, 5)
```

さなわれわれがのためにしたならば、たちのはにされるので、のためのもうつのはコンパイルされるだろう。そのため、この`.toIterable`またはしているコレクションにじてするをにのためにするとです。

オンラインでオプションクラスをむ https://riptutorial.com/ja/scala/topic/2293/オプションクラス

17: カッシング

- aFunction10_ // '_'をするりのパラメータグループのすべてのパラメータがカリングされることをコンパイラにします。
- nArityFunction.curried // n-arityをのカレーバージョンにします
- anotherFunctionx_Stringz//のパラメータをカリングします。それはにべられたをとする。

Examples

カルトとしてな

```
def multiply(factor: Int)(numberToBeMultiplied: Int): Int = factor * numberToBeMultiplied

val multiplyBy3 = multiply(3)_     // resulting function signature Int => Int
val multiplyBy10 = multiply(10)_ // resulting function signature Int => Int

val sixFromCurriedCall = multiplyBy3(2) //6
val sixFromFullCall = multiply(3)(2)    //6

val fortyFromCurriedCall = multiplyBy10(4) //40
val fortyFromFullCall = multiply(10)(4)    //40
```

なるタイプののパラメータグループ、ののカリングパラメータ

```
def numberOrCharacterSwitch(toggleNumber: Boolean)(number: Int)(character: Char): String =
  if (toggleNumber) number.toString else character.toString

// need to explicitly specify the type of the parameter to be curried
// resulting function signature Boolean => String
val switchBetween3AndE = numberOrCharacterSwitch(_: Boolean)(3)('E')

switchBetween3AndE(true) // "3"
switchBetween3AndE(false) // "E"
```

のパラメータグループでをカリングする

```
def minus(left: Int, right: Int) = left - right

val numberMinus5 = minus(_: Int, 5)
val fiveMinusNumber = minus(5, _: Int)

numberMinus5(7)    //  2
fiveMinusNumber(7) // -2
```

カッシング

2つののをしましょう

```
def add: (Int, Int) => Int = (x,y) => x + y
val three = add(1,2)
```

Currying `add`は、 **1**つの `Int`をとり、 を **1**つの `Int`から`Int` すにします。

```
val addCurried: (Int) => (Int => Int) = add2.curried
//               ^~~ take *one* Int
//                        ^~~~ return a *function* from Int to Int

val add1: Int => Int = addCurried(1)
val three: Int = add1(2)
val allInOneGo: Int = addCurried(1)(2)
```

このは、のをるにできます。のをとるをカリングすると、 **1**つのをるののアプリケーションにされます。

```
def add3: (Int, Int, Int) => Int = (a,b,c) => a + b + c + d
def add3Curr: Int => (Int => (Int => Int)) = add3.curried

val x = add3Curr(1)(2)(42)
```

カッシング

Wikipediaによると 、カリングは、

> のをるのをののにするです。

には、スカラーのから、2つのをる2があるのでは、

```
val f: (A, B) => C // a function that takes two arguments of type `A` and `B` respectively
                   // and returns a value of type `C`
```

に

```
val curriedF: A => B => C // a function that take an argument of type `A`
                          // and returns *a function*
                          // that takes an argument of type `B` and returns a `C`
```

したがって、arity-2の、カリーをのようにくことができます。

```
def curry[A, B, C](f: (A, B) => C): A => B => C = {
  (a: A) => (b: B) => f(a, b)
}
```

```
val f: (String, Int) => Double = {(_, _) => 1.0}
val curriedF: String => Int => Double = curry(f)
f("a", 1)        // => 1.0
curriedF("a")(1) // => 1.0
```

Scalaはこれについくつかのをしています

1. メソッドとしてカルトをくことができます。 `curriedF`はのようにくことができます

```
def curriedFAsAMethod(str: String)(int: Int): Double = 1.0
val curriedF = curriedFAsAMethod _
```

2. なライブラリ―メソッドをしてカレットをりくことができますつまり、 `A => B => C`から`(A, B) => C` `Function.uncurried`

```
val f: (String, Int) => Double = Function.uncurried(curriedF)
f("a", 1) // => 1.0
```

カッティングをする

currying は、のをとるのを、それぞれがのをつのシ―ケンスのにするです 。

これは、、のようなにです。

1. のなるはなるにされます 。 *1*
2. のなるは、アプリケ―ションのなるによってされます 。 *2*

1

はとボ―ナスでされたであるとしましょう

```
val totalYearlyIncome:(Int,Int) => Int =  (income, bonus) => income + bonus
```

2-arityのカレ―バ―ジョンはのとおりです。

```
val totalYearlyIncomeCurried: Int => Int => Int = totalYearlyIncome.curried
```

のでは、そのタイプはのように/することができます。

```
Int => (Int => Int)
```

がにかっているとしましょう。

```
val partialTotalYearlyIncome: Int => Int = totalYearlyIncomeCurried(10000)
```

そして、あるでは、ボ―ナスはかっています。

```
partialTotalYearlyIncome(100)
```

2

のにはとのがであるとしましょう。

```
val carManufacturing:(String,String) => String = (wheels, body) => wheels + body
```

これらのは、なるでされます。

```
class CarWheelsFactory {
  def applyCarWheels(carManufacturing:(String,String) => String): String => String =
          carManufacturing.curried("applied wheels..")
}

class CarBodyFactory {
  def applyCarBody(partialCarWithWheels: String => String): String =
partialCarWithWheels("applied car body..")
}
```

の`CarWheelsFactory`はのを`CarWheelsFactory`、のみをすることにして`CarWheelsFactory`。

のプロセスは、のをとります

```
val carWheelsFactory = new CarWheelsFactory()
val carBodyFactory   = new CarBodyFactory()

val carManufacturing:(String,String) => String = (wheels, body) => wheels + body

val partialCarWheelsApplied: String => String  =
carWheelsFactory.applyCarWheels(carManufacturing)
val carCompleted = carBodyFactory.applyCarBody(partialCarWheelsApplied)
```

Currying

たちがっているものは、クレジットカードのリストです。クレジットカードがうべきすべてのカードのをしたいといます。はクレジットカードのにするため、はそれにじてします。

たちはすでにのクレジットカードのプレミアムをし、がしたカードのをにれたをっています

```
case class CreditCard(creditInfo: CreditCardInfo, issuer: Person, account: Account)

object CreditCard {
  def getPremium(totalCards: Int, creditCard: CreditCard): Double = { ... }
}
```

、このにするなアプローチは、クレジットカードをプレミアムにマッピングし、それをにらすことです。このようなもの

```
val creditCards: List[CreditCard] = getCreditCards()
val allPremiums = creditCards.map(CreditCard.getPremium).sum //type mismatch; found : (Int,
CreditCard) ⇒ Double required: CreditCard ⇒ ?
```

しかし、`CreditCard.getPremium`は2つのパラメータをとするため、コンパイラはこれをにしません。へのなクレジットカードのをにし、そのをしてクレジットカードをにマッピングすることができます。なのは、`getPremium`をのパラメータリストをするようにしてカレーする`getPremium`です。

はのようになります。

```
object CreditCard {
  def getPremium(totalCards: Int)(creditCard: CreditCard): Double = { ... }
}

val creditCards: List[CreditCard] = getCreditCards()

val getPremiumWithTotal = CreditCard.getPremium(creditCards.length)_

val allPremiums = creditCards.map(getPremiumWithTotal).sum
```

オンラインでカッシングをむ https://riptutorial.com/ja/scala/topic/1636/カッシング

18: クラスとオブジェクト

- `class MyClass{} // curly braces are optional here as class body is empty`
- `class MyClassWithMethod {def method: MyClass = ???}`
- `new MyClass() //Instantiate`
- `object MyObject // Singleton object`
- `class MyClassWithGenericParameters[V1, V2](vl: V1, i: Int, v2: V2)`
- `class MyClassWithImplicitFieldCreation[V1](val v1: V1, val i: Int)`
- `new MyClassWithGenericParameters(2.3, 4, 5)`またはなるタイプの`new MyClassWithGenericParameters[Double, Any](2.3, 4, 5)`
- `class MyClassWithProtectedConstructor protected[my.pack.age](s: String)`

Examples

クラスインスタンスのインスタンス

Scalaのクラスは、クラスインスタンスの「」です。インスタンスには、そのクラスでされているとがまれます。クラスをするには

```
class MyClass{}  // curly braces are optional here as class body is empty
```

`new`キーワードをしてインスタンスをインスタンスすることができます。

```
var instance = new MyClass()
```

または

```
var instance = new MyClass
```

Scalaでは、のないコンストラクタをつクラスからオブジェクトをするためのカッコはオプションです。クラスコンストラクタがをる

```
class MyClass(arg : Int)       // Class definition
var instance = new MyClass(2)  // Instance instantiation
instance.arg                   // not allowed
```

ここで`MyClass`は、でのみできる`Int`が1つです。 `arg`は、フィールドとしてされていないり、`MyClass`からアクセスすることはできません

```
class MyClass(arg : Int){
    val prop = arg  // Class field declaration
}

var obj = new MyClass(2)
obj.prop     // legal statement
```

あるいは、コンストラクタでpublicとしてすることもできます。

```
class MyClass(val arg : Int)   // Class definition with arg declared public
var instance = new MyClass(2)  // Instance instantiation
instance.arg                   //arg is now visible to clients
```

パラメータなしでクラスをインスタンスする{} vs

コンストラクタをたないMyClassクラスがあるとしましょう

```
class MyClass
```

Scalaでは、のをしてインスタンスできます。

```
val obj = new MyClass()
```

あるいはにのようにくことができます

```
val obj = new MyClass
```

しかし、をわなければ、オプションのかっこでしないがじることがあります。のスレッドでするタスクをしたいとします。はサンプルコードです

```
val newThread = new Thread { new Runnable {
        override def run(): Unit = {
            // perform task
            println("Performing task.")
        }
      }
    }

newThread.start   // prints no output
```

このサンプルコードをすると`Performing task.`がされるとえられ`Performing task.`しかし、いたことに、それはもされません。ここでがこっているかてみましょう。よりしくおりになりたいは、`new Thread`に`{}`をしています。 `Thread`をするアノニマスクラスをしました

```
val newThread = new Thread {
  //creating anonymous class extending Thread
}
```

そして、このアノニマスクラスので、たちのタスクをしましたもう、 `Runnable`インターフェースをするアノニマスクラスをします。だから々は、々がすることをえているかもしれない`public Thread(Runnable target)`コンストラクタをにして`()`々が`public Thread()`のでされてもコンストラクタを`run()`メソッド。をするには、のわりにかっこをするがあります。

```
val newThread = new Thread ( new Runnable {
        override def run(): Unit = {
            // perform task
            println("Performing task.")
        }
```

```
      }
    )
```

いえれば、ここで`{}`と`()`はできません。

シングルトンコンパニオンオブジェクト

シングルトンオブジェクト

Scalaはメンバーをサポートしますが、Javaとじではありません。 Scalaはこれにわる*Singleton Objects*をします。シングルトンオブジェクトは、 `new`キーワードをしてインスタンスすることはできないをいて、のクラスにています。は、シングルトンクラスのサンプルです。

```
object Factorial {
    private val cache = Map[Int, Int]()
    def getCache = cache
}
```

「class」や「trait」ではなくsingletonオブジェクトをするために`object`キーワードをしてい`object`。シングルトンオブジェクトはインスタンスできないため、パラメータをつことはできません。シングルトンオブジェクトへのアクセスはのようになります。

```
Factorial.getCache() //returns the cache
```

これは、Javaクラスのメソッドにアクセスするのとまったくじようにえることにしてください。

コンパニオンオブジェクト

Scalaでは、シングルトンオブジェクトはするクラスのをできます。このようなシナリオでは、シングルトンオブジェクトはコンパニオンオブジェクトとばれます。たとえば、クラス`Factorial`がされ、そのにコンパニオンオブジェクト `Factorial`というがされています。により、コンパニオンオブジェクトはコンパニオンクラスとじファイルにされます。

```
class Factorial(num : Int) {

  def fact(num : Int) : Int = if (num <= 1) 1 else (num * fact(num - 1))

  def calculate() : Int = {
    if (!Factorial.cache.contains(num)) {    // num does not exists in cache
      val output = fact(num) // calculate factorial
      Factorial.cache += (num -> output)     // add new value in cache
    }

    Factorial.cache(num)
  }
}

object Factorial {
  private val cache = scala.collection.mutable.Map[Int, Int]()
```

```
}

val factfive = new Factorial(5)
factfive.calculate  // Calculates the factorial of 5 and stores it
factfive.calculate  // uses cache this time
val factfiveagain = new Factorial(5)
factfiveagain.calculate  // Also uses cache
```

このでは、プライベートcacheをして、のをするためにをしています。

ここでは、 object Factorialはコンパニオンオブジェクトであり、 class Factorialはするコンパニオンクラスです。コンパニオンオブジェクトとクラスは、おいのprivateメンバーにアクセスできます。のでは、 Factorialクラスは、そのコンパニオンオブジェクトのプライベートcacheメンバーにアクセスしています。

クラスのしいインスタンスでは、じコンパニオンオブジェクトがききされるため、そのオブジェクトのメンバーへのはきがれます。

オブジェクト

クラスはにていますが、オブジェクトはですつまり、すでにインスタンスされています。

```
object Dog {
    def bark: String = "Raf"
}

Dog.bark() // yields "Raf"
```

らはしばしばクラスのコンパニオンとしてされ、あなたはのようにくことができます

```
class Dog(val name: String) {

}

object Dog {
    def apply(name: String): Dog = new Dog(name)
}

val dog = Dog("Barky") // Object
val dog = new Dog("Barky") // Class
```

インスタンスのチェック

タイプチェック variable.isInstanceOf[Type]

パターンマッチング このではあまりにちません

```
variable match {
  case _: Type => true
  case _ => false
}
```

`isInstanceOf`とパターンマッチングでは、をいて、オブジェクトのだけをチェックし、パラメータをたないジェネリックパラメータはチェックしません。

```
val list: List[Any] = List(1, 2, 3)             //> list  : List[Any] = List(1, 2, 3)

val upcasting = list.isInstanceOf[Seq[Int]]     //> upcasting  : Boolean = true

val shouldBeFalse = list.isInstanceOf[List[String]]
                                                //> shouldBeFalse  : Boolean = true
```

しかし

```
val chSeqArray: Array[CharSequence] = Array("a") //> chSeqArray  : Array[CharSequence] =
Array(a)
val correctlyReified = chSeqArray.isInstanceOf[Array[String]]
                                               //> correctlyReified  : Boolean = false


val stringIsACharSequence: CharSequence = ""    //> stringIsACharSequence  : CharSequence = ""

val sArray = Array("a")                         //> sArray  : Array[String] = Array(a)
val correctlyReified = sArray.isInstanceOf[Array[String]]
                                                //> correctlyReified  : Boolean = true

//val arraysAreInvariantInScala: Array[CharSequence] = sArray
//Error: type mismatch;  found   : Array[String]  required: Array[CharSequence]
//Note: String <: CharSequence, but class Array is invariant in type T.
//You may wish to investigate a wildcard type such as `_ <: CharSequence`. (SLS 3.2.10)
//Workaround:
val arraysAreInvariantInScala: Array[_ <: CharSequence] = sArray
                                                //> arraysAreInvariantInScala  : Array[_ <:
CharSequence] = Array(a)


val arraysAreCovariantOnJVM = sArray.isInstanceOf[Array[CharSequence]]
                                                //> arraysAreCovariantOnJVM  : Boolean = true
```

キャスト `variable.asInstanceOf[Type]`

パターンマッチングでは

```
variable match {
  case _: Type => true
}
```

```
  val x = 3                                     //> x  : Int = 3
  x match {
    case _: Int => true//better: do something
    case _ => false
  }                                             //> res0: Boolean = true

  x match {
    case _: java.lang.Integer => true//better: do something
    case _ => false
  }                                             //> res1: Boolean = true
```

```
  x.isInstanceOf[Int]                             //> res2: Boolean = true

  //x.isInstanceOf[java.lang.Integer]//fruitless type test: a value of type Int cannot also be
a Integer

  trait Valuable { def value: Int}
  case class V(val value: Int) extends Valuable

  val y: Valuable = V(3)                          //> y  : Valuable = V(3)
  y.isInstanceOf[V]                               //> res3: Boolean = true
  y.asInstanceOf[V]                               //> res4: V = V(3)
```

これはJVMのにするもので、のプラットフォームJS、ネイティブのキャスト/チェックではがなるがあります。

コンストラクタ

プライマリコンストラクタ

Scalaでは、プライマリコンストラクタはクラスのです。クラスのには、コンストラクタであるパラメータリストがきます。 のとに、のパラメータリストはすることができます。

```
class Foo(x: Int, y: String) {
    val xy: String = y * x
    /* now xy is a public member of the class */
}

class Bar {
    ...
}
```

`val`キーワードでインスタンスメンバとしてマークされていないり、インスタンスのパラメータはコンストラクタのではアクセスできません。

```
class Baz(val z: String)
// Baz has no other members or methods, so the body may be omitted

val foo = new Foo(4, "ab")
val baz = new Baz("I am a baz")
foo.x // will not compile: x is not a member of Foo
foo.xy // returns "abababab": xy is a member of Foo
baz.z // returns "I am a baz": z is a member of Baz
val bar0 = new Bar
val bar1 = new Bar() // Constructor parentheses are optional here
```

オブジェクトのインスタンスがインスタンスされるときにされるは、クラスのにされます。

```
class DatabaseConnection
    (host: String, port: Int, username: String, password: String) {
    /* first connect to the DB, or throw an exception */
    private val driver = new AwesomeDB.Driver()
    driver.connect(host, port, username, password)
    def isConnected: Boolean = driver.isConnected
```

```
    ...
}
```

できるだけをなくすることはコンストラクタにくことをおめします。のコードのわりに、コードがIOのスケジューリングをするように、 `connect`と`disconnect`をする`connect`あり`disconnect`。

コンストラクタ

クラスには、コンストラクタとばれるのコンストラクタがあるかもしれません。これらは`def this(...) = e`というのコンストラクタでされ`def this(...) = e`ここで`e`はのコンストラクタをびさなければなりません

```
class Person(val fullName: String) {
  def this(firstName: String, lastName: String) = this(s"$firstName $lastName")
}

// usage:
new Person("Grace Hopper").fullName // returns Grace Hopper
new Person("Grace", "Hopper").fullName // returns Grace Hopper
```

これは、コンストラクタがなるをつことができることをします。

```
class Person private(val fullName: String) {
  def this(firstName: String, lastName: String) = this(s"$firstName $lastName")
}

new Person("Ada Lovelace") // won't compile
new Person("Ada", "Lovelace") // compiles
```

このようにして、コンシューマコードがクラスをインスタンスするをすることができます。

オンラインでクラスとオブジェクトをむ https://riptutorial.com/ja/scala/topic/2047/クラスとオブジェクト

19: ケースクラス

- caseクラスFoo//パラメータをたないcaseクラスはのリストをたなければなりません
- caseクラスFooa1A1、...、aNAN//フィールドa1 ... aNをつcaseクラスをする
- caseオブジェクトBar //シングルトンのcaseクラスをする

Examples

ケースクラスの

ケースクラスごとにでされるの1つは、オブジェクトののをチェックするのではなく、々のメンバーフィールドののをチェックする`equals`メソッドです。

のクラスでは

```
class Foo(val i: Int)
val a = new Foo(3)
val b = new Foo(3)
println(a == b)// "false" because they are different objects
```

ケースクラスの

```
case class Foo(i: Int)
val a = Foo(3)
val b = Foo(3)
println(a == b)// "true" because their members have the same value
```

されたコード

`case`をすると、Scalaコンパイラはクラスのコードをにします。でこのコードをするのはで、エラーのになります。のケースクラス

```
case class Person(name: String, age: Int)
```

...のコードがにされます

```
class Person(val name: String, val age: Int)
  extends Product with Serializable
{
  def copy(name: String = this.name, age: Int = this.age): Person =
    new Person(name, age)

  def productArity: Int = 2

  def productElement(i: Int): Any = i match {
    case 0 => name
    case 1 => age
    case _ => throw new IndexOutOfBoundsException(i.toString)
```

```
  }

  def productIterator: Iterator[Any] =
    scala.runtime.ScalaRunTime.typedProductIterator(this)

  def productPrefix: String = "Person"

  def canEqual(obj: Any): Boolean = obj.isInstanceOf[Person]

  override def hashCode(): Int = scala.runtime.ScalaRunTime._hashCode(this)

  override def equals(obj: Any): Boolean = this.eq(obj) || obj match {
    case that: Person => this.name == that.name && this.age == that.age
    case _ => false
  }

  override def toString: String =
    scala.runtime.ScalaRunTime._toString(this)
}
```

`case`は、コンパニオンオブジェクトもします。

```
object Person extends AbstractFunction2[String, Int, Person] with Serializable {
  def apply(name: String, age: Int): Person = new Person(name, age)

  def unapply(p: Person): Option[(String, Int)] =
    if(p == null) None else Some((p.name, p.age))
}
```

`object`にすると、 `case`はそれほどではありませんがのをちます。ここでのなは、 `toString`と、プロセスでのある`hashCode`です。とのオブジェクトは、のをしくすることにしてください。

```
object Foo extends Product with Serializable {
  def productArity: Int = 0

  def productIterator: Iterator[Any] =
    scala.runtime.ScalaRunTime.typedProductIterator(this)

  def productElement(i: Int): Any =
    throw new IndexOutOfBoundsException(i.toString)

  def productPrefix: String = "Foo"

  def canEqual(obj: Any): Boolean = obj.isInstanceOf[this.type]

  override def hashCode(): Int = 70822 // "Foo".hashCode()

  override def toString: String = "Foo"
}
```

クラスとそのコンパニオンオブジェクトので`case`によってされるメソッドをですることは、としてです。

ケースクラスの

のクラスとして、ケースクラスのにはいくつかのがあります。

- すべてのコンストラクタは`public`あり、されたオブジェクトでアクセスできます、ここでしたようにそうではありません。

```
case class Dog1(age: Int)
val x = Dog1(18)
println(x.age) // 18 (success!)

class Dog2(age: Int)
val x = new Dog2(18)
println(x.age) // Error: "value age is not a member of Dog2"
```

- `toString` 、 `equals` 、 `hashCode` プロパティにづく、 `copy` 、 `apply` 、 `unapply`のメソッドのをします。

```
case class Dog(age: Int)
val d1 = Dog(10)
val d2 = d1.copy(age = 15)
```

- パターンマッチングのためのなメカニズムをします

```
sealed trait Animal // `sealed` modifier allows inheritance within current build-unit
only
case class Dog(age: Int) extends Animal
case class Cat(owner: String) extends Animal
val x: Animal = Dog(18)
x match {
    case Dog(x) => println(s"It's a $x years old dog.")
    case Cat(x) => println(s"This cat belongs to $x.")
}
```

ケースクラスと

Scalaコンパイラは、デフォルトでパラメータリストのすべてののに`val`ます。つまり、デフォルトでは、ケースクラスはです。パラメータにはアクセサメソッドがありますが、ミューテータメソッドはありません。えば

```
case class Foo(i: Int)

val fooInstance = Foo(1)
val j = fooInstance.i       // get
fooInstance.i = 2           // compile-time exception (mutation: reassignment to val)
```

caseクラスのパラメータを`var`としてすると、デフォルトのがオーバーライドされ、caseクラスがになります。

```
case class Bar(var i: Int)

val barInstance = Bar(1)
val j = barInstance.i       // get
```

```
barInstance.i = 2           // set
```

caseクラスが 'mutable'であるののは、caseクラスのがであるです。

```
import scala.collection._

case class Bar(m: mutable.Map[Int, Int])

val barInstance = Bar(mutable.Map(1 -> 2))
barInstance.m.update(1, 3)                  // mutate m
barInstance                                 // Bar(Map(1 -> 3)
```

ここでこっている「」は、 mすマップにあり、 mではないことにしてください。したがって、あるのオブジェクトがメンバとしてmをっていれば、もにえます。のでは、 instanceAをすると instanceAもされinstanceB。

```
import scala.collection.mutable

case class Bar(m: mutable.Map[Int, Int])

val m = mutable.Map(1 ->2)
val barInstanceA = Bar(m)
val barInstanceB = Bar(m)
barInstanceA.m.update(1,3)
barInstanceA  // Bar = Bar(Map(1 -> 3))
barInstanceB  // Bar = Bar(Map(1 -> 3))
m  // scala.collection.mutable.Map[Int,Int] = Map(1 -> 3)
```

のをえたオブジェクトのコピーをする

ケースクラスは、のをえながら、いフィールドとじフィールドをするしいオブジェクトをする copyメソッドをしcopy。

このをして、とじをつしいオブジェクトをすることができます。このをすシンプルなケースクラス

```
case class Person(firstName: String, lastName: String, grade: String, subject: String)
val putu = Person("Putu", "Kevin", "A1", "Math")
val mark = putu.copy(firstName = "Ketut", lastName = "Mark")
// mark: People = People(Ketut,Mark,A1,Math)
```

このでは、2つのオブジェクトがコピー firstNameとlastName にされているをいて、2つのオブジェクトがの grade = A1 、 subject = Math をしていることがわかります。

のためのケースクラス

のをするために、にはたちのドメインでプリミティブのをけたいとえています。たとえば、 name つPersonをしてみてください。、 nameをStringとしてエンコードします。ただし、 Personのnameをす Stringエラーメッセージをす Stringをさせることはしくありません。

```
def logError(message: ErrorMessage): Unit = ???
case class Person(name: String)
val maybeName: Either[String, String] = ??? // Left is error, Right is name
maybeName.foreach(logError) // But that won't stop me from logging the name as an error!
```

このようなとしをけるために、のようにデータをエンコードすることができます

```
case class PersonName(value: String)
case class ErrorMessage(value: String)
case class Person(name: PersonName)
```

`PersonName`と`ErrorMessage`、さらにはの`String`をぜると、たちのコードはコンパイルされません。

```
val maybeName: Either[ErrorMessage, PersonName] = ???
maybeName.foreach(reportError) // ERROR: tried to pass PersonName; ErrorMessage expected
maybeName.swap.foreach(reportError) // OK
```

しかしこれは、`PersonName`コンテナとので`String`をボックス/ボックスするがあるため、ランタイムオーバーヘッドがさくなります。これをけるために、`PersonName`クラスと`ErrorMessage`クラスをできます。

```
case class PersonName(val value: String) extends AnyVal
case class ErrorMessage(val value: String) extends AnyVal
```

オンラインでケースクラスをむ https://riptutorial.com/ja/scala/topic/1022/ケースクラス

20: コレクション

Examples

リストをソートする

のリストをすると、さまざまなをべえることができます。

```
val names = List("Kathryn", "Allie", "Beth", "Serin", "Alana")
```

`sorted()`のデフォルトのるいは、 `math.Ordering`をすること`math.Ordering`。 これは、の、 レクソグラフィックソートになります。

```
names.sorted
// results in: List(Alana, Allie, Beth, Kathryn, Serin)
```

`sortWith`すると、をしてのを`sortWith`できます。

```
names.sortWith(_.length < _.length)
// results in: List(Beth, Allie, Serin, Alana, Kathryn)
```

`sortBy`すると、をできます。

```
//A set of vowels to use
val vowels = Set('a', 'e', 'i', 'o', 'u')

//A function that counts the vowels in a name
def countVowels(name: String) = name.count(l => vowels.contains(l.toLower))

//Sorts by the number of vowels
names.sortBy(countVowels)
//result is: List(Kathryn, Beth, Serin, Allie, Alana)
```

リストやソートされたリストを `reverse

```
names.sorted.reverse
//results in: List(Serin, Kathryn, Beth, Allie, Alana)
```

リストはJavaメソッド`java.util.Arrays.sort`とそのScalaラッパー`scala.util.Sorting.quickSort`をしてソートすることもできます

```
java.util.Arrays.sort(data)
scala.util.Sorting.quickSort(data)
```

これらのメソッドは、コレクションのとアンボックス/ボクシングをけることができれば、きなコレクションをソートするときのパフォーマンスをさせることができます。パフォーマンスのいに

ついてのなは、 Scala Collection sorted、sortWith、sortBy Performanceをしてください 。

nのコピーをむリストをする

あるオブジェクト`x`コピーのコレクションをするには、 fillメソッドをします。このでは`List`しますが、これは`fill`がをつのコレクションでもえます

```
// List.fill(n)(x)
scala > List.fill(3)("Hello World")
res0: List[String] = List(Hello World, Hello World, Hello World)
```

リストとベクトルのチートシート

> のパフォーマンスがれているため、 `List`わりに`Vector`をするのがベストプラクティスです 。 パフォーマンスのについては、こちらをください 。 `Vector`どこにすることができます`List`されています。

リスト

```
List[Int]()          // Declares an empty list of type Int
List.empty[Int]      // Uses `empty` method to declare empty list of type Int
Nil                  // A list of type Nothing that explicitly has nothing in it

List(1, 2, 3)        // Declare a list with some elements
1 :: 2 :: 3 :: Nil   // Chaining element prepending to an empty list, in a LISP-style
```

をる

```
List(1, 2, 3).headOption // Some(1)
List(1, 2, 3).head       // 1

List(1, 2, 3).lastOption // Some(3)
List(1, 2, 3).last       // 3, complexity is O(n)

List(1, 2, 3)(1)         // 2, complexity is O(n)
List(1, 2, 3)(3)         // java.lang.IndexOutOfBoundsException: 4
```

のに

```
0 :: List(1, 2, 3)       // List(0, 1, 2, 3)
```

をする

```
List(1, 2, 3) :+ 4       // List(1, 2, 3, 4), complexity is O(n)
```

リスト

```
List(1, 2) ::: List(3, 4) // List(1, 2, 3, 4)
List.concat(List(1,2), List(3, 4)) // List(1, 2, 3, 4)
```

```
List(1, 2) ++ List(3, 4)  // List(1, 2, 3, 4)
```

な

```
List(1, 2, 3).find(_ == 3)                    // Some(3)
List(1, 2, 3).map(_ * 2)                      // List(2, 4, 6)
List(1, 2, 3).filter(_ % 2 == 1)              // List(1, 3)
List(1, 2, 3).fold(0)((acc, i) => acc + i * i) // 1 * 1 + 2 * 2 + 3 * 3 = 14
List(1, 2, 3).foldLeft("Foo")(_ + _.toString)  // "Foo123"
List(1, 2, 3).foldRight("Foo")(_ + _.toString) // "123Foo"
```

マップコレクションのチートシート

> これは`map`メソッドのコレクションとはなる`Map`のコレクションのをうことにしてください。

の

```
Map[String, Int]()
val m1: Map[String, Int] = Map()
val m2: String Map Int = Map()
```

マップは、ほとんどのの`tuples`コレクションとなすことができます。のはキーで、2のはです。

```
val l = List(("a", 1), ("b", 2), ("c", 3))
val m = l.toMap                               // Map(a -> 1, b -> 2, c -> 3)
```

をする

```
val m = Map("a" -> 1, "b" -> 2, "c" -> 3)

m.get("a")  // Some(1)
m.get("d")  // None
m("a")      // 1
m("d")      // java.util.NoSuchElementException: key not found: d

m.keys      // Set(a, b, c)
m.values    // MapLike(1, 2, 3)
```

をする

```
Map("a" -> 1, "b" -> 2) + ("c" -> 3)              // Map(a -> 1, b -> 2, c -> 3)
Map("a" -> 1, "b" -> 2) + ("a" -> 3)              // Map(a -> 3, b -> 2)
Map("a" -> 1, "b" -> 2) ++ Map("b" -> 3, "c" -> 4) // Map(a -> 1, b -> 3, c -> 4)
```

な

マップのがする `map` 、 `find` 、 `forEach`などでは、コレクションのは`tuples`です。 functionパラメータは、タプルアクセサ `_1` 、 `_2` 、またはcaseブロックをむをできます。

```
m.find(_._1 == "a")  // Some((a,1))
m.map {
  case (key, value) => (value, key)
}                    // Map(1 -> a, 2 -> b, 3 -> c)
m.filter(_._2 == 2)  // Map(b -> 2)
m.foldLeft(0){
  case (acc, (key, value: Int)) => acc + value
}                    // 6
```

コレクションのとフィルター

コレクションの「マッピング」は、 `map`をして、そのコレクションのをのでします。なはのとおりです。

```
val someFunction: (A) => (B) = ???
collection.map(someFunction)
```

あなたはをすることができます

```
collection.map((x: T) => /*Do something with x*/)
```

に2をける

```
// Initialize
val list = List(1,2,3)
// list: List[Int] = List(1, 2, 3)

// Apply map
list.map((item: Int) => item*2)
// res0: List[Int] = List(2, 4, 6)

// Or in a more concise way
list.map(_*2)
// res1: List[Int] = List(2, 4, 6)
```

フィルタ

`filter`は、コレクションののをまたはしたいときにされます。 `map`に、はをりますが、そのは`Boolean`さなければなりません

```
val someFunction: (a) => Boolean = ???
collection.filter(someFunction)
```

をすることができます

```
collection.filter((x: T) => /*Do something that returns a Boolean*/)
```

ペアの

```
val list = 1 to 10 toList
// list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

// Filter out all elements that aren't evenly divisible by 2
list.filter((item: Int) => item % 2==0)
// res0: List[Int] = List(2, 4, 6, 8, 10)
```

そののマップとフィルタの

```
case class Person(firstName: String,
                  lastName: String,
                  title: String)

// Make a sequence of people
val people = Seq(
  Person("Millie", "Fletcher", "Mrs"),
  Person("Jim", "White", "Mr"),
  Person("Jenny", "Ball", "Miss") )


// Make labels using map
val labels = people.map( person =>
  s"${person.title}. ${person.lastName}"
)

// Filter the elements beginning with J
val beginningWithJ = people.filter(_.firstName.startsWith("J"))

// Extract first names and concatenate to a string
val firstNames = people.map(_.firstName).reduce( (a, b) => a + "," + b )
```

Scalaコレクションの

Scala Collectionsフレームワークは、 そのによれば 、いやすく、で、で、く、にされています。

フレームワークは、コレクションをするためのビルディングブロックとしてされたScalaのでされています。これらのビルディングブロックのについては、Scalaコレクションのをんでください 。

これらのビルトインコレクションは、なパッケージとなパッケージにかれています。デフォルトでは、のバージョンがされます。 `List()` もインポートすることなくをすると、 のリストがされます。

このフレームワークのもなの1つは、じえのコレクションでしたいやすいインターフェイスです。たとえば、コレクションのすべてのをすることは、リスト、セット、ベクトル、およびでじです 。

```
val numList = List[Int](1, 2, 3, 4, 5)
numList.reduce((n1, n2) => n1 + n2)  // 15

val numSet = Set[Int](1, 2, 3, 4, 5)
numSet.reduce((n1, n2) => n1 + n2)   // 15

val numArray = Array[Int](1, 2, 3, 4, 5)
numArray.reduce((n1, n2) => n1 + n2) // 15
```

これらののタイプは、 `Traversable`をしています。

> のパフォーマンスがれているため、 `List`わりに`Vector`をするのがベストプラクティスです 。 パフォーマンスのについては、こちらをごください 。 `Vector`どこにすることができます`List`されています。

トラバーサルタイプ

`Traversable` traitをつコレクションクラスは`foreach`をし、コレクションにしてのをするためのくのメソッドをします。これらのメソッドはすべてじようにします。もなはのとおりです。

- Map - `map` 、 `flatMap` 、および`collect`は、のコレクションのにをして、しいコレクション`collect`します。

```
List(1, 2, 3).map(num => num * 2) // double every number = List(2, 4, 6)

// split list of letters into individual strings and put them into the same list
List("a b c", "d e").flatMap(letters => letters.split(" ")) // = List("a", "b", "c", "d", "e")
```

- コンバージョン - `toList` 、 `toArray`などのコンバージョンでは、のコレクションをよりののコレクションにします。これらは、、 'to'でまるメソッドとよりなですつまり、 'toList'は`List`されます。

```
val array: Array[Int] = List[Int](1, 2, 3).toArray // convert list of ints to array of ints
```

- サイズ - `isEmpty` 、 `nonEmpty` 、 `size` 、および`hasDefiniteSize`は、セットにするすべてのメタデータです。これにより、コレクションのきがになります。また、コードがコレクションのサイズかかなどをめることができます。

```
List().isEmpty // true
List(1).nonEmpty // true
```

- - `head` 、 `last` 、 `find` 、およびその`Option`バリアントは、またはのをするために、またはコレクションののをつけるためにされます。

```
val list = List(1, 2, 3)
list.head // = 1
list.last // = 3
```

- サブコレクション - `filter` 、 `tail` 、 `slice` 、 `drop`などのにより、コレクションのをしてさらにできます。

```
List(-2, -1, 0, 1, 2).filter(num => num > 0) // = List(1, 2)
```

- `partition` - `partition` 、 `splitAt` 、 `span` 、および`groupBy`は、のコレクションをなるにします。

```
// split numbers into < 0 and >= 0
List(-2, -1, 0, 1, 2).partition(num => num < 0) // = (List(-2, -1), List(0, 1, 2))
```

- テスト - `exists` 、 `forall`および`count`は、このコレクションをチェックしてをたしているかどうかをべるためにされるです。

```
List(1, 2, 3, 4).forall(num => num > 0) // = true, all numbers are positive
List(-3, -2, -1, 1).forall(num => num < 0) // = false, not all numbers are negative
```

- Folds - `foldLeft` `/:` `foldLeft` 、 `foldRight` `:\` 、 `reduceLeft` 、および`reduceRight`は、コレクションのするにバイナリをするためにされます。 りみのについて はこちらへ、のについてはここにしてください 。

りたたむ

`fold`メソッドは、のアキュムレータをし、をしてアキュムレータをにするをして、コレクションにしてをいます。

```
val nums = List(1,2,3,4,5)
var initialValue:Int = 0;
var sum = nums.fold(initialValue){
  (accumulator,currentElementBeingIterated) => accumulator + currentElementBeingIterated
}
println(sum) //prints 15 because 0+1+2+3+4+5 = 15
```

のでは、 `fold()`がされています。また、2つのをるきをうこともできます。これをのにれて、のをのようにきすことができます

```
def sum(x: Int, y: Int) = x+ y
val nums = List(1, 2, 3, 4, 5)
var initialValue: Int = 0
val sum = nums.fold(initialValue)(sum)
println(sum) // prints 15 because 0 + 1 + 2 + 3 + 4 + 5 = 15
```

をするとにします

```
initialValue = 2;
sum = nums.fold(initialValue){
 (accumulator,currentElementBeingIterated) => accumulator + currentElementBeingIterated
}
println(sum) //prints 17 because 2+1+2+3+4+5 = 17
```

foldメソッドにはfoldLeftとfoldRight 2つのバリエーションがありfoldRight。

foldLeft()はからへコレクションののからそののまでfoldLeft()りします。 foldRight()、からへのからのまでfoldRight()りします。 fold() foldLeft()ようにからにします。、 fold()はfoldLeft()にびします。

```
def fold[A1 >: A](z: A1)(op: (A1, A1) => A1): A1 = foldLeft(z)(op)
```

fold()、 foldLeft()およびfoldRight()は、とじのをします。しかし、とはなりfoldLeft()とfoldRight()するためにえられ、fold()だけじタイプまたはコレクションのタイプのスーパータイプのものとすることができます。

このでは、はありません。したがって、 fold()をfoldLeft()またはfoldRight()とはわりません。になをすると、がされます。

があるは、 foldLeft()よりfoldLeft()をfoldRight()してfoldRight()。 foldRight()はパフォーマンスがします。

フォアハ

foreachはコレクションイテレータのではをさないでしいです。わりに、のみをつにをします。えば

```
scala> val x = List(1,2,3)
x: List[Int] = List(1, 2, 3)

scala> x.foreach { println }
1
2
3
```

foreachれるはのりをつことができますが、はされます。、 foreachは、がましいにされます。データをするは、 map、 filter、 for comprehensionのfor comprehension、またはのオプションをすることをしてください。

のの

```
def myFunc(a: Int) : Int = a * 2
List(1,2,3).foreach(myFunc) // Returns nothing
```

らす

reduce()、 reduceLeft()、およびreduceRightメソッドは、りみにています。 reduceにされるは2つのをとり、3つのをします。リストをする、の2つのはリストのの2つのです。のとリストののは、にされ、しいがされます。このしいは、リストののにされ、がなくなるまできます。がされます。

```
val nums = List(1,2,3,4,5)
sum = nums.reduce({ (a, b) => a + b })
println(sum) //prints 15

val names = List("John","Koby", "Josh", "Matilda", "Zac", "Mary Poppins")

def findLongest(nameA:String, nameB:String):String = {
  if (nameA.length > nameB.length) nameA else nameB
}

def findLastAlphabetically(nameA:String, nameB:String):String = {
  if (nameA > nameB) nameA else nameB
}

val longestName:String = names.reduce(findLongest(_,_))
println(longestName) //prints Mary Poppins

//You can also omit the arguments if you want
val lastAlphabetically:String = names.reduce(findLastAlphabetically)
println(lastAlphabetically) //prints Zac
```

りみとして、がどのようにするかにはいくつかのいがあります。らです

1. reduceにはアキュムレータはありません。
2. をリストでびすことはできません。
3. Reduceは、リストのまたはスーパータイプのみをすことができます。

オンラインでコレクションをむ https://riptutorial.com/ja/scala/topic/686/コレクション

21: シンボルリテラル

じじをつ2つのシンボルが、にしで、にじオブジェクトをします、つまりされている- Scalaは、シンボルのがしています。

シンボルはLisp、Ruby、Erlangなど、くののですが、Scalaではさなです。それにもかかわらずい。

つかいます

をつのリテラルに、つのがくが、、またはアンダースコア_リテラルシンボルです。のはではありえないのでです。

い

```
'ATM
'IPv4
'IPv6
'map_to_operations
'data_format_2006

// Using the `Symbol.apply` method

Symbol("hakuna matata")
Symbol("To be or not to be that is a question")
```

い

```
'8'th_division
'94_pattern
'bad-format
```

Examples

caseのの

データベース、ファイル、プロンプト 、およびリストをむのデータソースがあるとします 。されたソースにじて、たちのアプローチをします

```
def loadData(dataSource: Symbol): Try[String] = dataSource match {
  case 'database => loadDatabase() // Loading data from database
  case 'file =>  loadFile() // Loading data from file
  case 'prompt => askUser() // Asking user for data
  case 'argumentList => argumentListExtract() // Accessing argument list for data
  case _ => Failure(new Exception("Unsupported data source"))
}
```

たちは、 `Symbol`わりに`String`をよくうことができました。このでは、ののどれもにちませんので

、たちはしませんでした。

これにより、コーードがになり、エラーがしにくくなります。

オンラインでシンボルリテラルをむ https://riptutorial.com/ja/scala/topic/6419/シンボルリテラル

22: スカラズ

き

Scalazはプログラミングのためのscalaライブラリです。

これは、scalaライブラリのものをするになデータをします。それは、クラス `Functor` 、 `Monad` と、のデータにするインスタンスのセットをします。

Examples

ApplyUsage

```
import scalaz._
import Scalaz._

scala> Apply[Option].apply2(some(1), some(2))((a, b) => a + b)
res0: Option[Int] = Some(3)

scala> val intToString: Int => String = _.toString

scala> Apply[Option].ap(1.some)(some(intToString))
res1: Option[String] = Some(1)

scala> Apply[Option].ap(none)(some(intToString))
res2: Option[String] = None

scala> val double: Int => Int = _ * 2

scala> Apply[List].ap(List(1, 2, 3))(List(double))
res3: List[Int] = List(2, 4, 6)

scala> :kind Apply
scalaz.Apply's kind is X[F[A]]
```

FunctorUsage

```
import scalaz._
import Scalaz._
scala> val len: String => Int = _.length
len: String => Int = $$Lambda$1164/969820333@7e758f40

scala> Functor[Option].map(Some("foo"))(len)
res0: Option[Int] = Some(3)

scala> Functor[Option].map(None)(len)
res1: Option[Int] = None

scala> Functor[List].map(List("qwer", "adsfg"))(len)
res2: List[Int] = List(4, 5)
```

```
scala> :kind Functor
scalaz.Functor's kind is X[F[A]]
```

ArrowUsage

```
import scalaz._
import Scalaz._
scala> val plus1 = (_: Int) + 1
plus1: Int => Int = $$Lambda$1167/11131196496a6bfd97

scala> val plus2 = (_: Int) + 2
plus2: Int => Int = $$Lambda$1168/9243295486bbe050f

scala> val rev = (_: String).reverse
rev: String => String = $$Lambda$1227/1278001332@72685b74

scala> plus1.first apply (1, "abc")
res0: (Int, String) = (2,abc)

scala> plus1.second apply ("abc", 2)
res1: (String, Int) = (abc,3)

scala> rev.second apply (1, "abc")
res2: (Int, String) = (1,cba)

scala> plus1 *** rev apply(7, "abc")
res3: (Int, String) = (8,cba)

scala> plus1 &&& plus2 apply 7
res4: (Int, Int) = (8,9)

scala> plus1.product apply (1, 2)
res5: (Int, Int) = (2,3)

scala> :kind Arrow
scalaz.Arrow's kind is X[F[A1,A2]]
```

オンラインでスカラズをむ https://riptutorial.com/ja/scala/topic/9893/スカラズ

23: スカラドック

- メソッド、フィールド、クラスまたはパッケージのにします。
- `/**`まる
- には、コメントきでまる`*`があります
- わり`*/`

パラメータ

パラメータ	
クラス	–
`@constructor detail`	クラスのなコンストラクタをします。
メソッド	–
`@return detail`	メソッドでされるの。
メソッド、コンストラクタ、クラスタグ	–
`@param x detail`	メソッドまたはコンストラクタのvalueパラメータ$_x$。
`@tparam x detail`	メソッドまたはコンストラクタのパラメータ$_x$。
`@throws detail`	がになるか。
	–
`@see detail`	のをしてください。
`@note detail`	または、またはのすべきまたはにするメモをします。
`@example detail`	サンプルコードまたはするサンプルドキュメントをします。
`@usecase detail`	なメソッドがすぎるやノイズがいに、されたメソッドをします。
その	–
`@author detail`	のにするをします。
`@version detail`	このがするバージョンをします。

パラメータ	
`@deprecated detail`	のエンティティをとマークします。

Examples

シンプルなScaladocからメソッドへ

```
/**
  * Explain briefly what method does here
  * @param x Explain briefly what should be x and how this affects the method.
  * @param y Explain briefly what should be y and how this affects the method.
  * @return Explain what is returned from execution.
  */
def method(x: Int, y: String): Option[Double] = {
  // Method content
}
```

オンラインでスカラドックをむ https://riptutorial.com/ja/scala/topic/4518/スカラドック

24: ストリーム

ストリームはされ、ジェネレータをするためにできることをします。ジェネレータは、のにではなく、されたタイプのしいアイテムをオンデマンドでまたはします。これにより、なだけがにわれます。

Examples

ストリームをしたランダムシーケンスの

`genRandom`は、びされるたびにするが4の1のストリームをします。

```
def genRandom: Stream[String] = {
  val random = scala.util.Random.nextFloat()
  println(s"Random value is: $random")
  if (random < 0.25) {
    Stream.empty[String]
  } else {
    ("%.3f : A random number" format random) #:: genRandom
  }
}

lazy val randos = genRandom  // getRandom is lazily evaluated as randos is iterated through

for {
  x <- randos
} println(x) // The number of times this prints is effectively randomized.
```

*lazily recurses*である`#::`にしてください。これは、のをストリームにするため、されるまでストリームのりのをしません。

によるストリーム

ストリームをしてににになるストリームをすることができます。

```
// factorial
val fact: Stream[BigInt] = 1 #:: fact.zipWithIndex.map{case (p,x)=>p*(x+1)}
fact.take(10)  // (1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880)
fact(24)       // 620448401733239439360000

// the Fibonacci series
val fib: Stream[BigInt] = 0 #:: fib.scan(1:BigInt)(_+_)
fib.take(10)  // (0, 1, 1, 2, 3, 5, 8, 13, 21, 34)
fib(124)      // 36726740705505779255899443

// random Ints between 10 and 99 (inclusive)
def rndInt: Stream[Int] = (util.Random.nextInt(90)+10) #:: rndInt
rndInt.take(10)  // (20, 95, 14, 44, 42, 78, 85, 24, 99, 85)
```

このでは、 Var、Val、Defのいはい。 `def`として、はされるたびにされます。 `val`としてはされ、

されたにされます。これは、でをすることでできます。

```
// def with extra output per calculation
def fact: Stream[Int] = 1 #:: fact.zipWithIndex.map{case (p,x)=>print("!");p*(x+1)}
fact(5)  // !!!!!!!!!!!!!! 120
fact(4)  // !!!!!!!!!! 24
fact(7)  // !!!!!!!!!!!!!!!!!!!!!!!!!!!! 5040

// now as val
val fact: Stream[Int] = 1 #:: fact.zipWithIndex.map{case (p,x)=>print("!");p*(x+1)}
fact(5)  // !!!!! 120
fact(4)  // 24
fact(7)  // !! 5040
```

これは、なぜ`Stream`が`val`としてしないのかをします。

```
val rndInt: Stream[Int] = (util.Random.nextInt(90)+10) #:: rndInt
rndInt.take(5)  // (79, 79, 79, 79, 79)
```

のストリーム

```
// Generate stream that references itself in its evaluation
lazy val primes: Stream[Int] =
  2 #:: Stream.from(3, 2)
    .filter { i => primes.takeWhile(p => p * p <= i).forall(i % _ != 0) }
    .takeWhile(_ > 0) // prevent overflowing

// Get list of 10 primes
assert(primes.take(10).toList == List(2, 3, 5, 7, 11, 13, 17, 19, 23, 29))

// Previously calculated values were memoized, as shown by toString
assert(primes.toString == "Stream(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, ?)")
```

オンラインでストリームをむ https://riptutorial.com/ja/scala/topic/3702/ストリーム

25: セルフタイプ

- タイプ{selfId => / のメンバーは、することができ`selfIdthis`か / }
- trait Type {selfIdOtherType => / *のメンバーは`selfId`をすることができ、 `OtherType` * /
- traitタイプ{selfIdOtherType1とOtherType2 => / * `selfId`は`OtherType1`と`OtherType2`タイプ `OtherType2` * /

ケーキのパターンでよくされます。

Examples

なタイプの

タイプは、それがされるなクラスのをするために、やクラスでできます。このシンタックスをして、 `this`ためにのをすることもできますオブジェクトをオブジェクトからするがあるにです。

いくつかのオブジェクトをしたいとします。そのためには、ストレージのインターフェイスをし、コンテナにをします。

```
trait Container[+T] {
  def add(o: T): Unit
}

trait PermanentStorage[T] {
  /* Constraint on self type: it should be Container
   * we can refer to that type as `identifier`, usually `this` or `self`
   * or the type's name is used. */
  identifier: Container[T] =>

  def save(o: T): Unit = {
    identifier.add(o)
    //Do something to persist too.
  }
}
```

このように、これらはじオブジェクトにはありませんが、 `Container`することなく`PermanentStorage`をすることはできません。

オンラインでセルフタイプをむ https://riptutorial.com/ja/scala/topic/4639/セルフタイプ

26: タイプパラメータGenerics

Examples

オプションの

パラメータされたのいは、 Optionです。にのようなですにされているいくつかのメソッドがあります。

```
sealed abstract class Option[+A] {
  def isEmpty: Boolean
  def get: A

  final def fold[B](ifEmpty: => B)(f: A => B): B =
    if (isEmpty) ifEmpty else f(this.get)

  // lots of methods...
}

case class Some[A](value: A) extends Option[A] {
  def isEmpty = false
  def get = value
}

case object None extends Option[Nothing] {
  def isEmpty = true
  def get = throw new NoSuchElementException("None.get")
}
```

これには、 Bのものをすパラメータされたメソッドfoldがあることもわかります。

パラメータされたメソッド

メソッドのりのは、パラメータのによってなります。このでは、 xパラメータであり、 Aのである x パラメータとしてられています、。

```
def f[A](x: A): A = x

f(1)          // 1
f("two")      // "two"
f[Float](3)   // 3.0F
```

Scalaはをしてりのをします。このは、パラメータでびされるメソッドをします。したがって、がです。 *はすべてのタイプAにしてされていないため、はコンパイルエラーです。

```
def g[A](x: A): A = 2 * x  // Won't compile
```

ジェネリックコレクション

Intsのリストをする

```
trait IntList { ... }

class Cons(val head: Int, val tail: IntList) extends IntList { ... }

class Nil extends IntList { ... }
```

ブーール、ダブルなどのリストをするがあるはどうなりますか

ジェネリックリストの

```
trait List[T] {
  def isEmpty: Boolean
  def head: T
  def tail: List[T]
}

class Cons[T](val head: [T], val tail: List[T]) extends List[T] {
  def isEmpty: Boolean = false
}

class Nil[T] extends List[T] {
  def isEmpty: Boolean = true

  def head: Nothing = throw NoSuchElementException("Nil.head")

  def tail: Nothing = throw NoSuchElementException("Nil.tail")
}
```

オンラインでタイプパラメータGenericsをむ https://riptutorial.com/ja/scala/topic/782/タイプパラメータ-generics-

27: タイプレベルプログラミング

Examples

タイプレベルプログラミング

リストのがであるがのをつのリストをえると、のをすることなくリストのにしてしてをできることがましいかもしれない。のでは、なリストにしてマッピングをしています。

がするため、できるのはらかののにされているので、をする。な`type Apply[A]`をつ`Projection`ののを`type Apply[A]`、 `def apply[A](a: A): Apply[A]`のをするために`def apply[A](a: A): Apply[A]`ます。

```
trait Projection {
  type Apply[A] // <: Any
  def apply[A](a: A): Apply[A]
}
```

`type Apply[A]`では、レベルではなくレベルでプログラミングしています。

たちのリストタイプは、のタイプとに、のによってパラメータされた`map`をし`map`。マップのはクラスであり、クラスとをすることによってなりますが、`HList`なければなりません。

```
sealed trait HList {
  type Map[P <: Projection] <: HList
  def map[P <: Projection](p: P): Map[P]
}
```

のリストである`HNil`の、のはにそれです。ここでは、し`trait HNil`たちがくことができるようにの`HNil`するわりに、タイプとして`HNil.type`

```
sealed trait HNil extends HList
case object HNil extends HNil {
  type Map[P <: Projection] = HNil
  def map[P <: Projection](p: P): Map[P] = HNil
}
```

`HCons`は、でないのリストです。ここでは、マップをするとき、のヘッドタイプは、ヘッド`P#Apply[H]` へののP#Apply[H]からじるものであり、のテールタイプは、 `HList`であることがられているテール `T#Map[P]` の`HList`

```
case class HCons[H, T <: HList](head: H, tail: T) extends HList {
  type Map[P <: Projection] = HCons[P#Apply[H], T#Map[P]]
  def map[P <: Projection](p: P): Map[P] = HCons(p.apply(head), tail.map(p))
}
```

そのようなは、いくつかののラッピングをすることです。のは、 `HCons[Option[String],`

`HCons[Option[Int], HNil]]`インスタンスをします。

```
HCons("1", HCons(2, HNil)).map(new Projection {
  type Apply[A] = Option[A]
  def apply[A](a: A): Apply[A] = Some(a)
})
```

オンラインでタイプレベルプログラミングをむ https://riptutorial.com/ja/scala/topic/3738/タイプレベルプログラミング

28: タイプ

Examples

+はパラメータをとしてマークします。ここでは「 Producerは A でです。

```
trait Producer[+A] {
  def produce: A
}
```

タイプのパラメータは、「」タイプとえることができます。 Aをとしてマーク A と、 Producer[X] <: Producer[Y] が X <: Y していることが X <: Y ます。えば、 Producer[Cat] はな Producer[Animal] です。されたすべてのネコもなです。

パラメータは、にれません。たちがしているとして、のでは、コンパイルされません Co[Cat] <: Co[Animal] が、 Co[Cat] た def handle(a: Cat): Unit のうことができない Animal でとされる Co[Animal]

```
trait Co[+A] {
  def produce: A
  def handle(a: A): Unit
}
```

このをう1つのアプローチは、タイプパラメータでまれたパラメータをすることです。のでは、 B が A スーパータイプであることがわかります。したがって、 Option[X] の def getOrElse[B >: X](b: => B): B は X のスーパータイプをけれることができるということを X <: Y にして Option[X] <: Option[Y] - Option[Y] とされる Y のスーパータイプをむ

```
trait Option[+A] {
  def getOrElse[B >: A](b: => B): B
}
```

デフォルトでは、すべてのパラメータはです - えられた trait A[B] 、 " A は B でです"といます。これは、 A[Cat] と A[Animal] 2つのパラメータをえられた、 A[Cat] <: A[Animal] でも A[Cat] >: A[Animal] Cat と Animal にかかわらず。

は、このようなをするをし、がなままになるようにパラメータのにするをします。

コントラスト

-はのパラメータをします。ここでは、 " Handler は A です"とっています。

```
trait Handler[-A] {
  def handle(a: A): Unit
}
```

パラメータは、「」とえることができます。マーキング A そのアサートとして Handler[X] <:

`Handler[Y]`し`X >: Y`。えば、 `Handler[Animal]`はな`Handler[Cat]`であり、 `Handler[Animal]`はcatsもするがあるためです。

パラメ―タは、にれません。のは、 `Contra[Animal] <: Contra[Cat]`しているのでコンパイルされませんが、 `Contra[Animal]`は`def produce: Animal`っています`Contra[Cat]`にじてをすることはされていない`def produce: Animal`

```
trait Contra[-A] {
   def handle(a: A): Unit
   def produce: A
}
```

ただし、をにするために、はパラメ―タののをにさせます`Handler[Animal]`は`Handler[Cat]`より"より"であるとみなされます。

パラメ―タのメソッドをオ―バ―ロ―ドすることはであるため、このはにのをするにのみになります。のでは`ofCat`のりのがよりな`ofAnimal` 、 `ofAnimal`はされません。

```
implicit def ofAnimal: Handler[Animal] = ???
implicit def ofCat: Handler[Cat] = ???

implicitly[Handler[Cat]].handle(new Cat)
```

このは`scala.math.Ordering`されるであり、 `scala.math.Ordering`はパラメ―タ`T`であるとして。 1つのは、あなたのtypeclassをにすることと、されたのサブクラスにするイベントでのをパラメ―タすることです

```
trait Person
object Person {
  implicit def ordering[A <: Person]: Ordering[A] = ???
}
```

コレクションの

コレクションは、タイプ*でなので、スーパータイプがなは、サブタイプのコレクションをすことができます。

```
trait Animal { def name: String }
case class Dog(name: String) extends Animal

object Animal {
  def printAnimalNames(animals: Seq[Animal]) = {
    animals.foreach(animal => println(animal.name))
  }
}

val myDogs: Seq[Dog] = Seq(Dog("Curly"), Dog("Larry"), Dog("Moe"))

Animal.printAnimalNames(myDogs)
// Curly
// Larry
```

```
// Moe
```

それはのようにはえないかもしれませんが、 `Seq[Animal]`がそのパラメータでするここでは`Seq`というを`Seq[Dog]`がけれるというはけれられます。

*は、ライブラリのセット

の

1つのメソッドがのをけれるようにするもあります。これは、あなたがなで`T`をしたいが、としてそれがしているので、かもしれない。

```
trait LocalVariance[T]{
  /// ??? throws a NotImplementedError
  def produce: T = ???
  // the implicit evidence provided by the compiler confirms that S is a
  // subtype of T.
  def handle[S](s: S)(implicit evidence: S <:< T) = {
    // and we can use the evidence to convert s into t.
    val t: T = evidence(s)
    ???
  }
}

trait A {}
trait B extends A {}

object Test {
  val lv = new LocalVariance[A] {}

  // now we can pass a B instead of an A.
  lv.handle(new B {})
}
```

オンラインでタイプをむ https://riptutorial.com/ja/scala/topic/1651/タイプ

29: タプル

タプルのさを23にするのはなぜですか

タプルは、コンパイラによってオブジェクトとしてきえられます。コンパイラは`Tuple1`から`Tuple22`アクセスできます。こののは、によってされました。

なぜタプルのさは0からえるのですか

`Tuple0`は`Unit`します。

Examples

しいタプルをする

タプルは、2から22ののコレクションです。タプルは、かっこでできます。サイズ2タプル「ペア」ともばれますには、のがあります。

```
scala> val x = (1, "hello")
x: (Int, String) = (1,hello)
scala> val y = 2 -> "world"
y: (Int, String) = (2,world)
scala> val z = 3 → "foo"     //example of using U+2192 RIGHTWARD ARROW
z: (Int, String) = (3,foo)
```

`x`はサイズ2のタプルです。タプルのにアクセスするには、`._1`、`._22`します。たとえば、`x._1`をして`x`タプルののにアクセスできます。`x._2`は2のにアクセスします。もっとエレガントに、タプルをうことができます。

サイズ2のタプルをするためののは、に`(key -> value)`ペアであるMapでされ`(key -> value)`。

```
scala> val m = Map[Int, String](2 -> "world")
m: scala.collection.immutable.Map[Int,String] = Map(2 -> world)

scala> m + x
res0: scala.collection.immutable.Map[Int,String] = Map(2 -> world, 1 -> hello)

scala> (m + x).toList
res1: List[(Int, String)] = List((2,world), (1,hello))
```

マップのペアのはので、1がキーで、aがそのキーにけられたであることがわかります。

コレクションのタプル

タプルはコレクションでよくされますが、のでするがあります。たとえば、のタプルのリストがあるとします。

```
scala> val l = List(1 -> 2, 2 -> 3, 3 -> 4)
l: List[(Int, Int)] = List((1,2), (2,3), (3,4))
```

なタプル･アンパックをしてをにするのはなようです。

```
scala> l.map((e1: Int, e2: Int) => e1 + e2)
```

ただし、この、のエラーがします。

```
<console>:9: error: type mismatch;
 found   : (Int, Int) => Int
 required: ((Int, Int)) => ?
              l.map((e1: Int, e2: Int) => e1 + e2)
```

スカラは、このでタプルをにアンパックすることはできません。このをするには2つのがあります。に、アクセサ`_1`と`_2`をします。

```
scala> l.map(e => e._1 + e._2)
res1: List[Int] = List(3, 5, 7)
```

もう1つのオプションは、 `case`をしてパターンマッチングをしてタプルをアンパックすることです。

```
scala> l.map{ case (e1: Int, e2: Int) => e1 + e2}
res2: List[Int] = List(3, 5, 7)
```

これらのは、タプルのコレクションにされるにされます。

オンラインでタプルをむ https://riptutorial.com/ja/scala/topic/4971/タプル

30: パーサー

ParseResult Cases

`ParseResult`には3つのがあります

- のとするのについてのマーカーで、する。
- マッチがされたのにするマーカーをむ。この、パーサーはそのにバックトラックします。ここでは、がされます。
- エラー。をします。バックトラックやそれのはわれません。

Examples

な

```
import scala.util.parsing.combinator._

class SimpleParser extends RegexParsers {
  // Define a grammar rule, turn it into a regex, and apply it the input.
  def word: Parser[String] = """[A-Z][a-z]+""".r ^^ { _.toString }
}

object SimpleParser extends SimpleParser {
  val parseAlice = parse(word, "Alice went to Alamo Square.")
  val parseBarb = parse(word, "barb went Upside Down.")
}

//Successfully finds a match
println(SimpleParser.parseAlice)
//Fails to find a match
println(SimpleParser.parseBarb)
```

はのようになります。

```
[1.6] parsed: Alice
res0: Unit = ()

[1.1] failure: string matching regex `[A-Z][a-z]+' expected but `b' found

barb went Upside Down.
^
```

`Alice`の`[1.6]`は、のが$_1$にあり、するりののが$_6$でまることをしています。

オンラインでパーサーをむ https://riptutorial.com/ja/scala/topic/3730/パーサー

31: ハイブのユーザー

Examples

Apache Sparkのなハイブ UDF

```
import org.apache.spark.sql.functions._

// Create a function that uses the content of the column inside the dataframe
val code = (param: String) => if (param == "myCode") 1 else 0
// With that function, create the udf function
val myUDF = udf(code)
// Apply the udf to a column inside the existing dataframe, creating a dataframe with the
additional new column
val newDataframe = aDataframe.withColumn("new_column_name", myUDF(col(inputColumn)))
```

オンラインでハイブのユーザーをむ https://riptutorial.com/ja/scala/topic/8241/ハイブのユーザー

32: パターンマッチング

- セレクタマッチpartialFunction
- セレクタの/のリスト//これはのでもなです

パラメーター

パラメータ	
セレクタ	がパターンマッチングされている。
	caseとcaseリスト。

Examples

シンプルなパターンマッチ

のは、をのとするをしています。

```
def f(x: Int): String = x match {
  case 1 => "One"
  case 2 => "Two"
  case _ => "Unknown!"
}

f(2)  // "Two"
f(3)  // "Unknown!"
```

ライブデモ

_は、 フォールスルーまたはデフォルトのですが、ではありません。

```
def g(x: Int): String = x match {
  case 1 => "One"
  case 2 => "Two"
}

g(1)  // "One"
g(3)  // throws a MatchError
```

をげるのをけるために、デフォルトのケース `case _ => <do something>` をするのがここではのプログラミングのです。 caseクラスがしていれば、コンパイラがをするのにつことにしてください。されたをするユーザーのタイプのもじです。マッチがである、デフォルトのケースはないかもしれません

また、インラインでされていないとすることもできます。これらは、 したでなければなりません

。これは、のをするか、バッククォートをむかのいずれかによってされます。

`One` `two`のでされているか、のパラメータとしてされている

```
val One: Int = 1
val two: Int = 2
```

それらはのでることができます

```
def g(x: Int): String = x match {
  case One => "One"
  case `two` => "Two"
}
```

えばJavaのようなのプログラミングとはなり、はありません。 caseブロックがとすると、それがされ、がします。したがって、もなケースはのケースブロックでなければなりません。

```
def f(x: Int): String = x match {
  case _ => "Default"
  case 1 => "One"
}

f(5) // "Default"
f(1) // "Default"
```

したによるパターンマッチング

なパターンマッチングでは、されるは、みスコープのをシャドウします。によっては、みスコープのをするがあります。

のは、とタプルのリストをり、タプルのしいリストをします。がタプルの1つののとしてする、2のがインクリメントされます。リストにまだしないは、しいタプルがされます。

```
def tabulate(char: Char, tab: List[(Char, Int)]): List[(Char, Int)] = tab match {
  case Nil => List((char, 1))
  case (`char`, count) :: tail => (char, count + 1) :: tail
  case head :: tail => head :: tabulate(char, tail)
}
```

のは、メソッドの`char`がパターンマッチで "にたれているパターンマッチングをしています。つまり、 `tabulate('x', ...)`をびすと、のcaseはのようにされます。

```
case('x', count) => ...
```

Scalaは、ティックマークでられたをしたとしてします。じようにでまるもします。

Seqのパターンマッチング

コレクションののなをするには

```
def f(ints: Seq[Int]): String = ints match {
  case Seq() =>
      "The Seq is empty !"
  case Seq(first) =>
      s"The seq has exactly one element : $first"
  case Seq(first, second) =>
      s"The seq has exactly two elements : $first, $second"
  case  s @ Seq(_, _, _) =>
      s"s is a Seq of length three and looks like ${s}"  // Note individual elements are not
bound to their own names.
  case s: Seq[Int] if s.length == 4 =>
      s"s is a Seq of Ints of exactly length 4"  // Again, individual elements are not bound
to their own names.
  case _ =>
      "No match was found!"
}
```

ライブデモ

のをし、りをコレクションとしてするには

```
def f(ints: Seq[Int]): String = ints match {
  case Seq(first, second, tail @ _*) =>
      s"The seq has at least two elements : $first, $second. The rest of the Seq is $tail"
  case Seq(first, tail @ _*) =>
      s"The seq has at least one element : $first. The rest of the Seq is $tail"
  // alternative syntax
  // here of course this one will never match since it checks
  // for the same thing as the one above
  case first +: tail =>
      s"The seq has at least one element : $first. The rest of the Seq is $tail"
  case _ =>
      "The seq didn't match any of the above, so it must be empty"
}
```

に、をするためにすることができるのをいて、のとのパターンマッチングをうことができる。

`Nil`と`::`をしている、シーケンスにパターンマッチングするとしますが、 `List`してしないがじることにしてください。これをけるには、 `Seq( ...)`と`+:`にをします。

`WrappedArray`、 `Vector`などでは`::`をすることはできません。

```
scala> def f(ints:Seq[Int]) = ints match {
     | case h :: t => h
     | case _ => "No match"
     | }
f: (ints: Seq[Int])Any

scala> f(Array(1,2))
res0: Any = No match
```

そして`+:`

```
scala> def g(ints:Seq[Int]) = ints match {
     | case h+:t => h
```

```
     | case _ => "No match"
     | }
g: (ints: Seq[Int])Any

scala> g(Array(1,2).toSeq)
res4: Any = 1
```

ガードの

caseをifとみわせて、パターンマッチングになロジックをすることができます。

```
def checkSign(x: Int): String = {
    x match {
      case a if a < 0 => s"$a is a negative number"
      case b if b > 0 => s"$b is a positive number"
      case c => s"$c neither positive nor negative"
    }
}
```

あなたのガードがなマッチをらないようにすることがですコンパイラはしばしばこれをキャッチしません

```
def f(x: Option[Int]) = x match {
    case Some(i) if i % 2 == 0 => doSomething(i)
    case None    => doSomethingIfNone
}
```

に`MatchError`がスローされます。すべてのケースについてするか、ワイルドカードのとをするケースをするがあります。

```
def f(x: Option[Int]) = x match {
    case Some(i) if i % 2 == 0 => doSomething(i)
    case _ => doSomethingIfNoneOrOdd
}
```

ケースクラスによるパターンマッチング

すべてのcaseクラスは、パターンマッチングのにcaseクラスのメンバをするためにできるextractorをします。

```
case class Student(name: String, email: String)

def matchStudent1(student: Student): String = student match {
    case Student(name, email) => s"$name has the following email: $email" // extract name and
email
}
```

パターンマッチングのすべてののルールがされます。ガードとをしてマッチングをできます。

```
def matchStudent2(student: Student): String = student match {
    case Student("Paul", _) => "Matched Paul" // Only match students named Paul, ignore email
```

```
    case Student(name, _) if name == "Paul" => "Matched Paul" // Use a guard to match students
named Paul, ignore email
    case s if s.name == "Paul" => "Matched Paul" // Don't use extractor; use a guard to match
students named Paul, ignore email
    case Student("Joe", email) => s"Joe has email $email" // Match students named Joe, capture
their email
    case Student(name, email) if name == "Joe" => s"Joe has email $email" // use a guard to
match students named Joe, capture their email
    case Student(name, email) => s"$name has email $email." // Match all students, capture
name and email
}
```

オプションでのマッチング

Optionタイプでマッチングしている

```
def f(x: Option[Int]) = x match {
    case Some(i) => doSomething(i)
    case None    => doSomethingIfNone
}
```

これは`fold`、 `map`/`getOrElse`をうことととに`getOrElse`

```
def g(x: Option[Int]) = x.fold(doSomethingIfNone)(doSomething)
def h(x: Option[Int]) = x.map(doSomething).getOrElse(doSomethingIfNone)
```

パターンマッチング

がされたであるオブジェクトをパターンマッチングすると、Scalaはコンパイルにすべてのケースが「にしている」ことをチェックします

```
sealed trait Shape
case class Square(height: Int, width: Int) extends Shape
case class Circle(radius: Int) extends Shape
case object Point extends Shape


def matchShape(shape: Shape): String = shape match {
    case Square(height, width) => "It's a square"
    case Circle(radius)        => "It's a circle"
    //no case for Point because it would cause a compiler warning.
}
```

`Shape`しい`case class`がでされると、 `Shape`すべての`match`がコンパイラのをスローしめます。これにより、なリファクタリングがになります。コンパイラは、がなすべてのコードをにします。

Regexとのパターンマッチング

```
val emailRegex: Regex = "(.+)@(.+)\\.(.+)".r

"name@example.com" match {
  case emailRegex(userName, domain, topDomain) => println(s"Hi $userName from $domain")
```

```
  case _ => println(s"This is not a valid email.")
}
```

このでは、はされたメールアドレスにするようにみます。そうであれば、 `userName`と`domain`がされてされます。 `topDomain`もされますが、このではもわれません。 `str.r`びしは、 `new Regex(str)`とじです。 `r`は、 のによってできます。

パターンバインダー@

`@`は、パターンマッチングにをにバインドします。バインドされたは、したオブジェクトまたはしたオブジェクトののいずれかです。

```
sealed trait Shape
case class Rectangle(height: Int, width: Int) extends Shape
case class Circle(radius: Int) extends Shape
case object Point extends Shape

(Circle(5): Shape) match {
  case Rectangle(h, w) => s"rectangle, $h x $w."
  case Circle(r) if r > 9 => s"large circle"
  case c @ Circle(_) => s"small circle: ${c.radius}"  // Whole matched object is bound to c
  case Point => "point"
}
```

```
> res0: String = small circle: 5
```

バインドされたは、きフィルタでできます。って

```
case Circle(r) if r > 9 => s"large circle"
```

のようにくことができます

```
case c @ Circle(_) if c.radius > 9 => s"large circle"
```

は、するパターンのにのみバインドできます。

```
Seq(Some(1), Some(2), None) match {
  // Only the first element of the matched sequence is bound to the name 'c'
  case Seq(c @ Some(1), _*) => head
  case _ => None
}
```

```
> res0: Option[Int] = Some(1)
```

パターンマッチングタイプ

`isInstanceOf[B]`ではなく、パターンマッチングをしてインスタンスのタイプをチェックすることもできます。

```
val anyRef: AnyRef = ""
```

```
anyRef match {
  case _: Number       => "It is a number"
  case _: String       => "It is a string"
  case _: CharSequence => "It is a char sequence"
}
//> res0: String = It is a string
```

ケースのはです。

```
anyRef match {
  case _: Number       => "It is a number"
  case _: CharSequence => "It is a char sequence"
  case _: String       => "It is a string"
}
//> res1: String = It is a char sequence
```

このように、フォールスルーをたないな 'switch'ステートメントにています。しかし、のタイプからパターンマッチと "をみわせることもできます。えば

```
case class Foo(s: String)
case class Bar(s: String)
case class Woo(s: String, i: Int)

def matcher(g: Any):String = {
  g match {
    case Bar(s) => s + " is classy!"
    case Foo(_) => "Someone is wicked smart!"
    case Woo(s, _) => s + " is adventerous!"
    case _ => "What are we talking about?"
  }
}

print(matcher(Foo("Diana")))  // prints 'Diana is classy!'
print(matcher(Bar("Hadas")))  // prints 'Someone is wicked smart!'
print(matcher(Woo("Beth", 27)))   // prints 'Beth is adventerous!'
print(matcher(Option("Katie")))  // prints 'What are we talking about?'
```

`Foo`と`Woo`、アンダースコア _ をして 'unbound variableとさせる'ことにしてください。つまり、このは`Hadas`と`27` はにバインドされていないため、そののハンドラではできません。これは、そのがであるかをにせずに「の」にさせるためになです。

テーブルスイッチまたはルックアップスイッチとしてコンパイルされたパターンマッチング

`@switch`アノテーションは、バイトコードレベルで`match`ステートメントをの`tableswitch`にきえることができることをコンパイラにします。これは、になやのをりくことができるマイナーなです。

`@switch`アノテーションは、リテラルと`final val`とのにしてのみします。パターンマッチを`tableswitch` / `lookupswitch`としてコンパイルできない、コンパイラはをします。

```
import annotation.switch
```

```
def suffix(i: Int) = (i: @switch) match {
  case 1 => "st"
  case 2 => "nd"
  case 3 => "rd"
  case _ => "th"
}
```

はのパターンマッチとじです

```
scala> suffix(2)
res1: String = "2nd"

scala> suffix(4)
res2: String = "4th"
```

Scalaドキュメント 2.8から`@switch`

> にされる。する、コンパイラは、マッチがテーブルスイッチまたはルックアップスイッチにコンパイルされていることをし、わりにのにコンパイルするとエラーをします。

Javaから

- テーブルスイッチ「インデックスとジャンプによるジャンプテーブルへのアクセス」
- lookupswitch "キーマッチとジャンプによるジャンプテーブルへのアクセス"

にのパターンをさせる

|じをるためにのにして1つのcaseをさせるためにうことができます

```
def f(str: String): String = str match {
  case "foo" | "bar" => "Matched!"
  case _ => "No match."
}

f("foo")  // res0: String = Matched!
f("bar")  // res1: String = Matched!
f("fubar")  // res2: String = No match.
```

このでをマッチングするとうまくいくが、のタイプのマッチングによってがすることにしてください。

```
sealed class FooBar
case class Foo(s: String) extends FooBar
case class Bar(s: String) extends FooBar

val d = Foo("Diana")
val h = Bar("Hadas")

// This matcher WILL NOT work.
```

```
def matcher(g: FooBar):String = {
  g match {
    case Foo(s) | Bar(s) => print(s)  // Won't work: s cannot be resolved
    case Foo(_) | Bar(_) => _         // Won't work: _ is an unbound placeholder
    case _ => "Could not match"
  }
}
```

の _ 、バインドされていないのをとせず、のかをしたいは、くいく

```
def matcher(g: FooBar):String = {
  g match {
    case Foo(_) | Bar(_) => "Is either Foo or Bar."  // Works fine
    case _ => "Could not match"
  }
}
```

それのは、あなたのケースをしてします

```
def matcher(g: FooBar):String = {
  g match {
    case Foo(s) => s
    case Bar(s) => s
    case _ => "Could not match"
  }
}
```

タプルのパターンマッチング

のタプルの`List`をえられます

```
val pastries = List(("Chocolate Cupcake", 2.50),
                    ("Vanilla Cupcake", 2.25),
                    ("Plain Muffin", 3.25))
```

パターンマッチングをすると、を々にできます。

```
pastries foreach { pastry =>
  pastry match {
    case ("Plain Muffin", price) => println(s"Buying muffin for $price")
    case p if p._1 contains "Cupcake" => println(s"Buying cupcake for ${p._2}")
    case _ => println("We don't sell that pastry")
  }
}
```

のケースでは、のとしてするをするをしています。 2のケースでは、 タプルのとする ifおよび tupleのをしています。

オンラインでパターンマッチングをむ https://riptutorial.com/ja/scala/topic/661/パターンマッチング

33: パッケージ

き

Scalaのパッケージはきなプログラムのをします。たとえば、 `com.sql`と`org.http`パッケージでの`connection`がわれます。これらのパッケージにアクセスするには、`com.sql.connection`および`org.http.connection`をそれぞれできます。

Examples

パッケージ

```
package com {
    package utility {
        package serialization {
            class Serializer
            ...
        }
    }
}
```

パッケージとファイル

packageは、つかったファイルにバインドされません。パッケージのをのファイルでつけることができます。えば、のパッケージはmath1.scalaファイルとmath2.scalaファイルにあります。

math1.scalaファイル

```
package org {
    package math {
        package statistics {
            class Interval
        }
    }
}
```

ファイルmath2.scala

```
package org {
    package math{
        package probability {
            class Density
        }
    }
}
```

ファイルstudy.scala

```
import org.math.probability.Density
import org.math.statistics.Interval

object Study {

    def main(args: Array[String]): Unit = {
        var a = new Interval()
        var b = new Density()
    }
}
```

パッケージ

Scalaパッケージは、Javaパッケージのにうがあります。
クラスやインタフェースとのをけるため、パッケージはすべてでされています。は、のインターネットドメインをしてパッケージをします。たとえば、

```
io.super.math
```

オンラインでパッケージをむ https://riptutorial.com/ja/scala/topic/8231/パッケージ

34: ベストプラクティス

> val、なオブジェクト、およびのないメソッドをむ。にらにをばしてください。のとそのがあるは、のある、なオブジェクト、およびメソッドをします。

- Odersky、Spoon、Vennersによる*Scala*プログラミング

このプレゼンテーションでは 、Oderskyのとがあります。

Examples

にしないでおく

なタスクをにしないでください。ほとんどの、なのはのものだけです

- データ
-
- モナドのようなAPI `map` 、 `flatMap` 、 `fold`

Scalaにはのようななものがたくさんあります。

- のための`Cake pattern`または`Reader Monad`。
- `implicit`としてのをす。

これらのことは、にとってはではありません。するにそれらのをけてください。のがないなコンセプトをすると、コードをし、メンテナンスをさせます。

1つのにあまりにくのをしないでください。

- ののあるをつける。
- `for`または`map`にをします。

のようなものがあるとしましょう

```
if (userAuthorized.nonEmtpy) {
  makeRequest().map {
    case Success(respone) =>
      someProcessing(..)
      if (resendToUser) {
        sendToUser(...)
      }
    ...
  }
}
```

すべてのが`Either`またはの`Validation`ようなをすは、のようにくことができます

```
for {
  user     <- authorizeUser
  response <- requestToThirdParty(user)
  _        <- someProcessing(...)
} {
  sendToUser
}
```

になスタイルをむ

デフォルトでは

- であれば、 `var`ではなく`val`してください。これにより、をむくのユーティリティをシームレスにできます。
- `recursion`と`comprehensions`のではなく、ループを。
- なコレクションをします。これはなり`val`をするためのです。
- CRUDではなく、データ、CQRSスタイルのロジックにをてます。

スタイルをするのにはいがあります。

- `var`は、ローカルたとえば、アクターでできます。
- `mutable`のでれたパフォーマンスをします。

オンラインでベストプラクティスをむ https://riptutorial.com/ja/scala/topic/4376/ベストプラクティス

35: マクロ

き

マクロは、コンパイルのメタプログラミングのです。やメソッドなど、Scalaコードののは、コンパイルにのコードをするためにできます。マクロは、のコードをすデータでするのScalaコードです。 [Macro Paradise] []プラグインは、マクロのををえてします。 [マクロパラダイス] http://docs.scala-lang.org/overviews/macros/paradise.html

- def x=マクロx_impl // xはマクロで、x_implはコードをするためにわれます
- def macroTransformannotteesAny *Any =マクロimpl //アノテーションでマクロをする

マクロは、 `scala.language.macros`をインポート`scala.language.macros`か、コンパイラオプション`-language:macros`して、にするがあるです。マクロだけがこれをとします。マクロをするコードではありません。

Examples

マクロ

このなマクロは、きアイテムをそのままします。

```
import scala.annotation.{compileTimeOnly, StaticAnnotation}
import scala.reflect.macros.whitebox.Context

@compileTimeOnly("enable macro paradise to expand macro annotations")
class noop extends StaticAnnotation {
  def macroTransform(annottees: Any*): Any = macro linkMacro.impl
}

object linkMacro {
  def impl(c: Context)(annottees: c.Expr[Any]*): c.Expr[Any] = {
    import c.universe._

    c.Expr[Any](q"{..$annottees}")
  }
}
```

`@compileTimeOnly`アノテーションは、このマクロをするために`paradise`コンパイラプラグインをインクルードするがあることをすメッセージでエラーをします。 SBTをしてこれをめるはここにあります 。

のマクロは、のようにできます。

```
@noop
case class Foo(a: String, b: Int)
```

```
@noop
object Bar {
  def f(): String = "hello"
}

@noop
def g(): Int = 10
```

メソッドマクロ

メソッドがマクロであるとされている、コンパイラはとしてされたコードをけり、それをASTにします。に、そのASTをしてマクロをびし、しいASTをします。しいASTがコールサイトにスプライスされます。

```
import reflect.macros.blackbox.Context

object Macros {
  // This macro simply sees if the argument is the result of an addition expression.
  // E.g. isAddition(1+1) and isAddition("a"+1).
  // but !isAddition(1+1-1), as the addition is underneath a subtraction, and also
  // !isAddition(x.+), and !isAddition(x.+(a,b)) as there must be exactly one argument.
  def isAddition(x: Any): Boolean = macro isAddition_impl

  // The signature of the macro implementation is the same as the macro definition,
  // but with a new Context parameter, and everything else is wrapped in an Expr.
  def isAddition_impl(c: Context)(expr: c.Expr[Any]): c.Expr[Boolean] = {
    import c.universe._ // The universe contains all the useful methods and types
    val plusName = TermName("+").encodedName // Take the name + and encode it as $plus
    expr.tree match { // Turn expr into an AST representing the code in isAddition(...)
      case Apply(Select(_, `plusName`), List(_)) => reify(true)
      // Pattern match the AST to see whether we have an addition
      // Above we match this AST
      //             Apply (function application)
      //            /     \
      //         Select  List(_) (exactly one argument)
      // (selection ^ of entity, basically the . in x.y)
      //      /          \
      //    _              \
      //               `plusName` (method named +)
      case _                                    => reify(false)
      // reify is a macro you use when writing macros
      // It takes the code given as its argument and creates an Expr out of it
    }
  }
}
```

`Tree`をとしてるマクロをつこともです。 `Expr`をするために`reify`がどのようにされるかのように、`q` quasiquoteは`Tree`してすることができます。の`q`をすることもできたことにしてください `expr.tree`はく`expr.tree`ですが、 `Tree`そのものですが、ではありませんでした。

```
// No Exprs, just Trees
def isAddition_impl(c: Context)(tree: c.Tree): c.Tree = {
  import c.universe._
  tree match {
    // q is a macro too, so it must be used with string literals.
    // It can destructure and create Trees.
```

```
    // Note how there was no need to encode + this time, as q is smart enough to do it itself.
    case q"${_} + ${_}" => q"true"
    case _              => q"false"
  }
}
```

マクロのエラー

マクロは、 `Context`してコンパイラのとエラーをきこすがあります。

たちは、いコードについてはにだとし、コンパイラのメッセージでなのすべてのインスタンスをマークしたいとしますこのアイデアがどれほどいかについてはえません。このようなメッセージをすはもしないマクロをうことができます。

```
import reflect.macros.blackbox.Context

def debtMark(message: String): Unit = macro debtMark_impl
def debtMarkImpl(c: Context)(message: c.Tree): c.Tree = {
  message match {
    case Literal(Constant(msg: String)) => c.info(c.enclosingPosition, msg, false)
    // false above means "do not force this message to be shown unless -verbose"
    case _                              => c.abort(c.enclosingPosition, "Message must be a
string literal.")
    // Abort causes the compilation to completely fail. It's not even a compile error, where
    // multiple can stack up; this just kills everything.
  }
  q"()" // At runtime this method does nothing, so we return ()
}
```

さらに、 `???`をするわりにされていないコードをマークするために、々は、つのマクロをすることができます`!!!`と`?!?`じをたしますが、コンパイラのをします。 `?!?`がされ、 `!!!`なエラーがします。

```
import reflect.macros.blackbox.Context

def ?!? : Nothing = macro impl_?!?
def !!! : Nothing = macro impl_!!!

def impl_?!?(c: Context): c.Tree = {
  import c.universe._
  c.warning(c.enclosingPosition, "Unimplemented!")
  q"${termNames.ROOTPKG}.scala.Predef.???"
  // If someone were to shadow the scala package, scala.Predef.??? would not work, as it
  // would end up referring to the scala that shadows and not the actual scala.
  // ROOTPKG is the very root of the tree, and acts like it is imported anew in every
  // expression. It is actually named _root_, but if someone were to shadow it, every
  // reference to it would be an error. It allows us to safely access ??? and know that
  // it is the one we want.
}

def impl_!!!(c: Context): c.Tree = {
  import c.universe._
  c.error(c.enclosingPosition, "Unimplemented!")
  q"${termNames.ROOTPKG}.scala.Predef.???"
}
```

オンラインでマクロをむ https://riptutorial.com/ja/scala/topic/3808/マクロ

36: モナド

Examples

モナド

には、モナドは`flatMap`このコンテナをすると`unit`このコンテナをするの2つのでパックされた`F[_]`とされたのコンテナです。

ライブラリのには、 `List[T]` 、 `Set[T]` 、 `Option[T]`ます。

な

Monad `M`は、 `flatMap`と`unit` 2つのをつパラメトリック `M[T]`です。

```
trait M[T] {
  def flatMap[U](f: T => M[U]): M[U]
}

def unit[T](x: T): M[T]
```

これらのは、

1. `(m flatMap f) flatMap g = m flatMap (x => f(x) flatMap g)`
   つまり、シーケンスがされていないは、のでタームをすることができます。したがって、 `m`を`f`にし、 `g`にをすると、 `f`を`g`にし、そのに`m`をするのとじがられます。
2. の `unit(x) flatMap f == f(x)`
   つまり、 `f`をってフラット・マップされた`x`のモナドは、 `f`を`x`にすることとである。
3. `m flatMap unit == m`
   これは「」です。ユニットにしてフラットマップされたモナドは、それにするモナドをします。

```
val m = List(1, 2, 3)
def unit(x: Int): List[Int] = List(x)
def f(x: Int): List[Int] = List(x * x)
def g(x: Int): List[Int] = List(x * x * x)
val x = 1
```

1.

```
(m flatMap f).flatMap(g) == m.flatMap(x => f(x) flatMap g) //Boolean = true
//Left side:
List(1, 4, 9).flatMap(g) // List(1, 64, 729)
//Right side:
 m.flatMap(x => (x * x) * (x * x) * (x * x)) //List(1, 64, 729)
```

2. の

```
unit(x).flatMap(x => f(x)) == f(x)
List(1).flatMap(x => x * x) == 1 * 1
```

3. の

```
//m flatMap unit == m
m.flatMap(unit) == m
List(1, 2, 3).flatMap(x => List(x)) == List(1,2,3) //Boolean = true
```

コレクションは**Monads**です

コレクションのほとんどは、モナド `List[T]` 、 `Option[T]` 、またはモナドのようなもの `Either[T]` 、 `Future[T]` です。これらのコレクションは`for`の`for`にににに`flatMap`ことができますこれは、`flatMap`ののき`flatMap`。

```
val a = List(1, 2, 3)
val b = List(3, 4, 5)
for {
  i <- a
  j <- b
} yield(i * j)
```

はのものとじです

```
a flatMap {
  i => b map {
    j => i * j
  }
}
```

モナドはデ―タをし、そののにしかしないため、ここではのためにモナディックデ―タをにチェ―ンすることができます。

オンラインでモナドをむ https://riptutorial.com/ja/scala/topic/4112/モナド

37: のスタイルでデータをう

とのは、よりいラクダのにするがあります

のはラクダのにするがあります。つまり、メンバがfinalでで、パッケージオブジェクトまたはオブジェクトにしているは、となすことができます

メソッド、、は、ラクダのがさい

http：//docs.scala-lang.org/style/naming-conventions.html

このコンパイル

```
val (a,b) = (1,2)
// a: Int = 1
// b: Int = 2
```

しかしこれはしません

```
val (A,B) = (1,2)
// error: not found: value A
// error: not found: value B
```

Examples

valとvarだけではありません

valとvar

```
scala> val a = 123
a: Int = 123

scala> a = 456
<console>:8: error: reassignment to val
       a = 456

scala> var b = 123
b: Int = 123

scala> b = 321
b: Int = 321
```

- valはできません Java finalのように、されたらできません
- varリファレンスは、Javaでのなとしてりてです

およびなコレクション

```
  val mut = scala.collection.mutable.Map.empty[String, Int]
  mut += ("123" -> 123)
  mut += ("456" -> 456)
  mut += ("789" -> 789)

  val imm = scala.collection.immutable.Map.empty[String, Int]
  imm + ("123" -> 123)
  imm + ("456" -> 456)
  imm + ("789" -> 789)

  scala> mut
    Map(123 -> 123, 456 -> 456, 789 -> 789)

  scala> imm
    Map()

scala> imm + ("123" -> 123) + ("456" -> 456) + ("789" -> 789)
    Map(123 -> 123, 456 -> 456, 789 -> 789)
```

Scalaのライブラリは、のデータとなデータのをします。のデータが「」されるたびに、のコレクションをインプレースでするわりに、しいインスタンスがされます。コレクションのインスタンスは、のインスタンスとなをすることができます。

でなコレクションScalaの

しかし、はこのにをすることはできません

のは、として、2るピックアップしてみましょうMapとりMapのすべてのむmaとmb

```
def merge2Maps(ma: Map[String, Int], mb: Map[String, Int]): Map[String, Int]
```

のみは、 `for ((k, v) <- map)` をってマップのをし、らかのでマージされたマップをすことができます。

```
def merge2Maps(ma: ..., mb: ...): Map[String, Int] = {

  for ((k, v) <- mb) {
    ???
  }

}
```

このにのきはすぐにをします。 そのの**for**とされています 。これは、をforにはよりfor 。

```
// this:
for ((k, v) <- map) { ??? }

// is equivalent to:
map.foreach { case (k, v) => ??? }
```

なぜ々はをこさなければならないのか

foreachはにっています。たちがforeachでかこることをむたびに、 ""がです。この、var resultをするか、なデ―タをすることができます。

resultマップのとりつぶし

のは、しようmaとmbしているscala.collection.immutable.Map 、たちがすることができresultからをma

```
val result = mutable.Map() ++ ma
```

に、をするmbをりし、 maののののkeyがすでにするは、 mbできします。

```
mb.foreach { case (k, v) => result += (k -> v) }
```

な

これまでのところにうまくいきました。「なコレクションをするがあります。

```
def merge2Maps(ma: Map[String, Int], mb: Map[String, Int]): Map[String, Int] = {
  val result = scala.collection.mutable.Map() ++ ma
  mb.foreach { case (k, v) => result += (k -> v) }
  result.toMap // to get back an immutable Map
}
```

り

```
scala> merge2Maps(Map("a" -> 11, "b" -> 12), Map("b" -> 22, "c" -> 23))
  Map(a -> 11, b -> 22, c -> 23)
```

レスキュ―へのりみ

このシナリオでforeachをりくにはどうすればよいですかには、コレクションをりししてをする、オプションのをするには.foldLeftをすることができます。

```
def merge2Maps(ma: Map[String, Int], mb: Map[String, Int]): Map[String, Int] = {
  mb.foldLeft(ma) { case (result, (k, v)) => result + (k -> v) }
  // or more concisely mb.foldLeft(ma) { _ + _ }
}
```

この、たちの ""は、 .foldLeft zeroであるmaからまる.foldLeft。

らかに、こののは、りたたみにくのMapインスタンスをしてしていますが、それらのインスタンスはMapなクロ―ンではなく、のインスタンスとなデ―タをしていることにするがあります。

よりな

.foldLeftアプロ―チのようにであれば、を.foldLeftするがです。のデ―タをすることで、のがに

なります。

オンラインでのスタイルでデータをうをむ https://riptutorial.com/ja/scala/topic/6298/のスタイルでデータをう

38: コレクション

コレクションは、レベルののをすことによってプログラミングをにします。これにより、マルチコア･アーキテクチャをにできます。コレクションのには、 `ParArray` 、 `ParVector` 、 `mutable.ParHashMap` 、 `immutable.ParHashMap` 、 `ParRange`などがあり`ParRange`。なリストは、ドキュメントにあります 。

Examples

コレクションのと

コレクションからコレクションをするには、 `par`メソッドをびします。コレクションからコレクションをするには、 `seq`メソッドをびします。のでは、の`Vector`を`ParVector` 、び`ParVector` `Vector`にします。

```
scala> val vect = (1 to 5).toVector
vect: Vector[Int] = Vector(1, 2, 3, 4, 5)

scala> val parVect = vect.par
parVect: scala.collection.parallel.immutable.ParVector[Int] = ParVector(1, 2, 3, 4, 5)

scala> parVect.seq
res0: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3, 4, 5)
```

`par`メソッドをさせることができます。これにより、コレクションをコレクションにし、ちにアクションをすることができます。

```
scala> vect.map(_ * 2)
res1: scala.collection.immutable.Vector[Int] = Vector(2, 4, 6, 8, 10)

scala> vect.par.map(_ * 2)
res2: scala.collection.parallel.immutable.ParVector[Int] = ParVector(2, 4, 6, 8, 10)
```

これらのでは、にはのにけられ、その、のをとせずにがしたにされます。

とし

コレクションをのでするがあるは、コレクションをしないでください。

コレクションはにをします。つまり、すべてのがのにされ、なるプロセッサにされます。プロセッサは、のがっているをらない。 コレクションのがな、してされるはです。じコードを2するとなるがられます。

ががなの、されたコレクションのはになります。

```
scala> val list = (1 to 1000).toList
list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10...

scala> list.reduce(_ - _)
res0: Int = -500498

scala> list.reduce(_ - _)
res1: Int = -500498

scala> list.reduce(_ - _)
res2: Int = -500498

scala> val listPar = list.par
listPar: scala.collection.parallel.immutable.ParSeq[Int] = ParVector(1, 2, 3, 4, 5, 6, 7, 8,
9, 10...

scala> listPar.reduce(_ - _)
res3: Int = -408314

scala> listPar.reduce(_ - _)
res4: Int = -422884

scala> listPar.reduce(_ - _)
res5: Int = -301748
```

`foreach`などのをうは、のためにされたコレクションではましくされないことがあります。これをするには、 `reduce`や`map`などののないをし`map`。

```
scala> val wittyOneLiner = Array("Artificial", "Intelligence", "is", "no", "match", "for",
"natural", "stupidity")

scala> wittyOneLiner.foreach(word => print(word + " "))
Artificial Intelligence is no match for natural stupidity

scala> wittyOneLiner.par.foreach(word => print(word + " "))
match natural is for Artificial no stupidity Intelligence

scala> print(wittyOneLiner.par.reduce{_ + " " + _})
Artificial Intelligence is no match for natural stupidity

scala> val list = (1 to 100).toList
list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15...
```

オンラインでコレクションをむ https://riptutorial.com/ja/scala/topic/3882/コレクション

39:

Examples

クラスをつCakeパターン。

```
//create a component that will be injected
trait TimeUtil {
    lazy val timeUtil = new TimeUtilImpl()

    class TimeUtilImpl{
        def now() = new DateTime()
    }
}

//main controller is depended on time util
trait MainController {
    _ : TimeUtil => //inject time util into main controller

    lazy val mainController = new MainControllerImpl()

    class MainControllerImpl {
        def printCurrentTime() = println(timeUtil.now()) //timeUtil is injected from TimeUtil
trait
    }
}

object MainApp extends App {
    object app extends MainController
       with TimeUtil //wire all components

    app.mainController.printCurrentTime()
}
```

のでは、MainControllerにTimeUtilをするをしました。

もなは、 `TimeUtil`を`MainController`にするための `_: TimeUtil => MainController`。 つまり、`MainController`は`MainController`し`TimeUtil`。

のえでは、クラスをすることができるようにテストするがだから、コンポーネントでクラス `TimeUtilImpl` をします。また、プロジェクトがよりになるときにメソッドがびされるをトレースすることもです。

に、すべてのコンポーネントをまとめてします。あなたがGuiceにれているなら、これは`Binding`します

オンラインでをむ https://riptutorial.com/ja/scala/topic/5909/

40:

Examples

をる

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

object FutureDivider {
    def divide(a: Int, b: Int): Future[Int] = Future {
        // Note that this is integer division.
        a / b
    }
}
```

にえば、 `divide`は、とオーバー`a b`。

をさせる

したをするもなは、のをることです。 `map`メソッドをすることです。いくつかのコードが`FutureDivider`オブジェクトの`divide`メソッドを "Creating a Future"のから`FutureDivider`ます。コードはのするようにえるためにはがでしょうオーバー`a b`

```
object Calculator {
    def calculateAndReport(a: Int, b: Int) = {
        val eventualQuotient = FutureDivider divide(a, b)

        eventualQuotient map {
            quotient => println(quotient)
        }
    }
}
```

したをする

のでがされることがあり、これによりFutureがすることがあります。 "Creating a future"のでは、びしコードが`55`と`0`を`divide`メソッドにすとどうなりますかもちろん、0でしようとすると`ArithmeticException`がスローされます。それはするコードでどのようにされますかには、にするがいくつかあります。

`recover`とパターンマッチングでをします。

```
object Calculator {
    def calculateAndReport(a: Int, b: Int) = {
        val eventualQuotient = FutureDivider divide(a, b)

        eventualQuotient recover {
```

```
            case ex: ArithmeticException => println(s"It failed with: ${ex.getMessage}")
        }
    }
}
```

が`failed`をします。ここではのになります。

```
object Calculator {
    def calculateAndReport(a: Int, b: Int) = {
        val eventualQuotient = FutureDivider divide(a, b)

        // Note the use of the dot operator to get the failed projection and map it.
        eventualQuotient.failed.map {
            ex => println(s"It failed with: ${ex.getMessage}")
        }
    }
}
```

をにする

のでは、とのケースをするFutureの々のをしました。しかし、、のははるかににされます。ここにがあります、もっとなやりでかれています

```
object Calculator {
    def calculateAndReport(a: Int, b: Int) = {
        val eventualQuotient = FutureDivider divide(a, b)

        eventualQuotient map {
            quotient => println(s"Quotient: $quotient")
        } recover {
            case ex: ArithmeticException => println(s"It failed with: ${ex.getMessage}")
        }
    }
}
```

シーケンシングとトラバース

によっては、々のでをするがあります。 `List[Future[Int]]`をつとしますが、わりに`List[Int]`をするがあります。それからは、 `List[Future[Int]]`このインスタンスを`Future[List[Int]]`です。こののために、 `Future`コンパニオンオブジェクトには`sequence`メソッドがあります。

```
def listOfFuture: List[Future[Int]] = List(1,2,3).map(Future(_))
def futureOfList: Future[List[Int]] = Future.sequence(listOfFuture)
```

な`sequence`は、 `F[G[T]]`を`F`と`G`して`G[F[T]]`するプログラミングのでにられているです。

`traverse`とばれるがありますが、のをしますが、なとしてをります。 `x => x`というをパラメータとしてすると、 `sequence`のようにします。

```
def listOfFuture: List[Future[Int]] = List(1,2,3).map(Future(_))
def futureOfList: Future[List[Int]] = Future.traverse(listOfFuture)(x => x)
```

しかし、なは、されたlistOfFutureののインスタンスをすることをにします。さらに、のはFutureリストであるはありません。したがって、このをのようにすることができます。

```
def futureOfList: Future[List[Int]] = Future.traverse(List(1,2,3))(Future(_))
```

この、 List(1,2,3)はのとしてされ、x => xはFuture(_)にきえられ、にIntをFutureにラップします。これのは、パフォーマンスをさせるためにList[Future[Int]]をできることです。

のをみわせる - のために

*for*のは、ののしたにするコードのブロックをするコンパクトなです。

f1, f2, f3 3つのFuture[String]それぞれone, two, threeのがまれます。

```
val fCombined =
    for {
        s1 <- f1
        s2 <- f2
        s3 <- f3
    } yield (s"$s1 - $s2 - $s3")
```

fCombinedは、すべてのがにすると、one - two - threeをむFuture[String]なります。

のExectionContextがここでされることにしてください。

また、のためには、flatMapメソッドのなるのなので、bodyのFutureオブジェクトは、でまれたコードブロックのをし、シーケンシャルコードにつながることにしてください。あなたはそれをにています

```
val result1 = for {
  first <- Future {
    Thread.sleep(2000)
    System.currentTimeMillis()
  }
  second <- Future {
    Thread.sleep(1000)
    System.currentTimeMillis()
  }
} yield first - second

val fut1 = Future {
  Thread.sleep(2000)
  System.currentTimeMillis()
}
val fut2 = Future {
  Thread.sleep(1000)
  System.currentTimeMillis()
}
val result2 = for {
  first <- fut1
  second <- fut2
} yield first - second
```

`result1`オブジェクトでまれたはにであり、 `result2`はであります。

なと`yield`については、 http//docs.scala-lang.org/tutorials/FAQ/yield.htmlをしてください。

オンラインでをむ https://riptutorial.com/ja/scala/topic/3245/

41:

Examples

テーール

のをすると、びしはのエントリをびしスタックにプッシュします。がすると、アプリケーションはエントリをポップダウンしてらなければなりません。くのなびしがある、それはなスタックでわるがあります。

スカラは、びしをにつけたににをします。 `@tailrec` をにして、テーールコールのをにうことができます。コンパイラは、をできない、エラーメッセージをします。

な

このは、びしがわれたときに、がびしがしたにとがあるをするがあるため、ではありません。

```
def fact(i : Int) : Int = {
   if(i <= 1) i
   else i * fact(i-1)
}

println(fact(5))
```

パラメータをしてをびすと、スタックはのようになります。

```
(fact 5)
(* 5 (fact 4))
(* 5 (* 4 (fact 3)))
(* 5 (* 4 (* 3 (fact 2))))
(* 5 (* 4 (* 3 (* 2 (fact 1)))))
(* 5 (* 4 (* 3 (* 2 (* 1 (fact 0))))))
(* 5 (* 4 (* 3 (* 2 (* 1 * 1)))))
(* 5 (* 4 (* 3 (* 2))))
(* 5 (* 4 (* 6)))
(* 5 (* 24))
120
```

このを`@tailrec`アノテーションしようとすると、のエラーメッセージがされます。 `could not optimize @tailrec annotated method fact: it contains a recursive call not in tail position`

テーール

では、まずをしてから、びしをして、のステップのをのステップにします。

```
def fact_with_tailrec(i : Int) : Long = {
   @tailrec
```

```
   def fact_inside(i : Int, sum: Long) : Long = {
      if(i <= 1) sum
      else fact_inside(i-1,sum*i)
   }
   fact_inside(i,1)
}

println(fact_with_tailrec(5))
```

に、テールのスタックトレースはのようになります。

```
(fact_with_tailrec 5)
(fact_inside 5 1)
(fact_inside 4 5)
(fact_inside 3 20)
(fact_inside 2 60)
(fact_inside 1 120)
```

`fact_inside`へのびしごとにじのデータをするがあるのは、がにをにすためです。つまり、`fact_with_tail 1000000`がびされたとしても、`fact_with_tail 3`とじのスペースしかとしません。これは、テールびしではてはまりません。そのようなきなはスタックオーバーフローをきこすがあります。

トランポリンによるスタックレス`scala.util.control.TailCalls`

のびしに`StackOverflowError`エラーがすることはにです。 Scalaライブラリは、ヒープオブジェクトとをしてスタックのオーバーフローをし、のローカルをするTailCallをします。

TailCallsのスカードからの2つの

```
import scala.util.control.TailCalls._

def isEven(xs: List[Int]): TailRec[Boolean] =
  if (xs.isEmpty) done(true) else tailcall(isOdd(xs.tail))

def isOdd(xs: List[Int]): TailRec[Boolean] =
  if (xs.isEmpty) done(false) else tailcall(isEven(xs.tail))

// Does this List contain an even number of elements?
isEven((1 to 100000).toList).result

def fib(n: Int): TailRec[Int] =
  if (n < 2) done(n) else for {
    x <- tailcall(fib(n - 1))
    y <- tailcall(fib(n - 2))
  } yield (x + y)

// What is the 40th entry of the Fibonacci series?
fib(40).result
```

オンラインでをむ https://riptutorial.com/ja/scala/topic/3889/

42:

Scalaのにはいくつかのがあるため、 `sealed trait case objects`と`case objects`をするアプローチがされます。

1. はもじです。
2. コンパイラは「マッチはではありません」とをっていません。と`scala.MatchError`ないとにします`scala.MatchError`

```
def isWeekendWithBug(day: WeekDays.Value): Boolean = day match {
  case WeekDays.Sun | WeekDays.Sat => true
}

isWeekendWithBug(WeekDays.Fri)
scala.MatchError: Fri (of class scala.Enumeration$Val)
```

とべて

```
def isWeekendWithBug(day: WeekDay): Boolean = day match {
  case WeekDay.Sun | WeekDay.Sat => true
}

Warning: match may not be exhaustive.
It would fail on the following inputs: Fri, Mon, Thu, Tue, Wed
def isWeekendWithBug(day: WeekDay): Boolean = day match {
                                              ^
```

Scala Enumerationのについては、このでしくしています 。

Examples

Scala Enumerationをした

Enumerationをすることによって、Javaのようなをできます 。

```
object WeekDays extends Enumeration {
  val Mon, Tue, Wed, Thu, Fri, Sat, Sun = Value
}

def isWeekend(day: WeekDays.Value): Boolean = day match {
  case WeekDays.Sat | WeekDays.Sun => true
  case _ => false
}

isWeekend(WeekDays.Sun)
res0: Boolean = true
```

のにがなをすることもできます。

```
object WeekDays extends Enumeration {
      val Mon = Value("Monday")
      val Tue = Value("Tuesday")
      val Wed = Value("Wednesday")
      val Thu = Value("Thursday")
      val Fri = Value("Friday")
      val Sat = Value("Saturday")
      val Sun = Value("Sunday")
}

println(WeekDays.Mon)
>> Monday

WeekDays.withName("Monday") == WeekDays.Mon
>> res0: Boolean = true
```

なるがじインスタンスとしてされるような、どおりではないにしてください。

```
object Parity extends Enumeration {
   val Even, Odd = Value
}

WeekDays.Mon.isInstanceOf[Parity.Value]
>> res1: Boolean = true
```

されたオブジェクトとケースオブジェクトの

`Enumeration`をするわりに、 `sealed`ケースオブジェクトをします。

```
sealed trait WeekDay

object WeekDay {
  case object Mon extends WeekDay
  case object Tue extends WeekDay
  case object Wed extends WeekDay
  case object Thu extends WeekDay
  case object Fri extends WeekDay
  case object Sun extends WeekDay
  case object Sat extends WeekDay
}
```

`sealed`キーワードは、`WeekDay`がのファイルでできないことをします。これにより、 `WeekDay`すべてのながにされていることをめて、コンパイラはのをできます。

1つのは、このメソッドではすべてのなのリストをできないということです。このようなリストをするには、にするがあります。

```
val allWeekDays = Seq(Mon, Tue, Wed, Thu, Fri, Sun, Sat)
```

ケースクラスは`sealed`をすることもできます。したがって、オブジェクトとケースクラスをしてなをすることができます。

```
sealed trait CelestialBody
```

```
object CelestialBody {
  case object Earth extends CelestialBody
  case object Sun extends CelestialBody
  case object Moon extends CelestialBody
  case class Asteroid(name: String) extends CelestialBody
}
```

もうつのは、 `sealed`オブジェクトののにアクセスする、またはそれによってするがないことです。にけられたがなは、でするがあります。

```
  sealed trait WeekDay { val name: String }

  object WeekDay {
      case object Mon extends WeekDay { val name = "Monday" }
      case object Tue extends WeekDay { val name = "Tuesday" }
      (...)
  }
```

あるいはに

```
  sealed case class WeekDay(name: String)

  object WeekDay {
      object Mon extends WeekDay("Monday")
      object Tue extends WeekDay("Tuesday")
      (...)
  }
```

されたとケースオブジェクトと**allValues-macro**の

これは、マクロがコンパイルにすべてのインスタンスをむセットをするされたのなるです。これは、がにをできますが、それをallElementsセットにすることをれてしまうというをいています。

このは、にきなのにです。

```
import EnumerationMacros._

sealed trait WeekDay
object WeekDay {
  case object Mon extends WeekDay
  case object Tue extends WeekDay
  case object Wed extends WeekDay
  case object Thu extends WeekDay
  case object Fri extends WeekDay
  case object Sun extends WeekDay
  case object Sat extends WeekDay
  val allWeekDays: Set[WeekDay] = sealedInstancesOf[WeekDay]
}
```

これをうには、のマクロがです。

```
import scala.collection.immutable.TreeSet
```

```
import scala.language.experimental.macros
import scala.reflect.macros.blackbox

/**
A macro to produce a TreeSet of all instances of a sealed trait.
Based on Travis Brown's work:
http://stackoverflow.com/questions/13671734/iteration-over-a-sealed-trait-in-scala
CAREFUL: !!! MUST be used at END OF code block containing the instances !!!
*/
object EnumerationMacros {
  def sealedInstancesOf[A]: TreeSet[A] = macro sealedInstancesOf_impl[A]

  def sealedInstancesOf_impl[A: c.WeakTypeTag](c: blackbox.Context) = {
    import c.universe._

    val symbol = weakTypeOf[A].typeSymbol.asClass

    if  (!symbol.isClass || !symbol.isSealed)
      c.abort(c.enclosingPosition, "Can only enumerate values of a sealed trait or class.")
    else {

      val children = symbol.knownDirectSubclasses.toList

      if (!children.forall(_.isModuleClass)) c.abort(c.enclosingPosition, "All children must
be objects.")
      else c.Expr[TreeSet[A]] {

        def sourceModuleRef(sym: Symbol) =
Ident(sym.asInstanceOf[scala.reflect.internal.Symbols#Symbol
          ].sourceModule.asInstanceOf[Symbol]
        )

        Apply(
          Select(
            reify(TreeSet).tree,
            TermName("apply")
          ),
          children.map(sourceModuleRef(_))
        )
      }
    }
  }
}
```

オンラインでをむ https://riptutorial.com/ja/scala/topic/1499/

43: びし

き

Scalaでは、メソッドをびしたり、オブジェクトのフィールドにアクセスしたりするときに、びしをすることができます。これをにくみむのではなく、マーカー[`scala.Dynamic`] [Dynamic scaladoc]によってにされるなとのルールをきすことでします。これにより、などにするオブジェクトにプロパティをにするをエミュレートできます。 [dynamic scaladoc]http://www.scala-lang.org/api/2.12.x/scala/Dynamic.html

- Fooクラスはにされています
- foo.field
- foo.field = value
- foo.methodargs
- foo.methodきArg = x、y

`Dynamic`サブタイプをするには、 `-language:dynamics`インポート`scala.language.dynamics`か、 `-language:dynamics`コンパイラオプションをして、の`dynamics`にするがあります。この`Dynamic`ユーザーは、のサブタイプをしていないため、これをにするはありません。

Examples

フィールドアクセス

この

```
class Foo extends Dynamic {
  // Expressions are only rewritten to use Dynamic if they are not already valid
  // Therefore foo.realField will not use select/updateDynamic
  var realField: Int = 5
  // Called for expressions of the type foo.field
  def selectDynamic(fieldName: String) = ???
  def updateDynamic(fieldName: String)(value: Int) = ???
}
```

フィールドへのなアクセスをにします

```
val foo: Foo = ???
foo.realField // Does NOT use Dynamic; accesses the actual field
foo.realField = 10 // Actual field access here too
foo.unrealField // Becomes foo.selectDynamic(unrealField)
foo.field = 10  // Becomes foo.updateDynamic("field")(10)
foo.field = "10" // Does not compile; "10" is not an Int.
foo.x() // Does not compile; Foo does not define applyDynamic, which is used for methods.
foo.x.apply() // DOES compile, as Nothing is a subtype of () => Any
// Remember, the compiler is still doing static type checks, it just has one more way to
// "recover" and rewrite otherwise invalid code now.
```

メソッドびし

この

```
class Villain(val minions: Map[String, Minion]) extends Dynamic {
  def applyDynamic(name: String)(jobs: Task*) = jobs.foreach(minions(name).do)
  def applyDynamicNamed(name: String)(jobs: (String, Task)*) = jobs.foreach {
    // If a parameter does not have a name, and is simply given, the name passed as ""
    case ("", task) => minions(name).do(task)
    case (subsys, task) => minions(name).subsystems(subsys).do(task)
  }
}
```

きパラメ―タのにかかわらず、メソッドへのびしをします。

```
val gru: Villain = ???
gru.blu() // Becomes gru.applyDynamic("blu")()
// Becomes gru.applyDynamicNamed("stu")(("fooer", ???), ("boomer", ???), ("", ???),
//          ("computer breaker", ???), ("fooer", ???))
// Note how the `???` without a name is given the name ""
// Note how both occurrences of `fooer` are passed to the method
gru.stu(fooer = ???, boomer = ???, ???, `computer breaker` = ???, fooer = ???)
gru.ERR("a") // Somehow, scalac thinks "a" is not a Task, though it clearly is (it isn't)
```

フィ―ルドアクセスとメソッドとの

ややにだけでなく、それをさせるためのい、これは

```
val dyn: Dynamic = ???
dyn.x(y) = z
```

のものとです。

```
dyn.selectDynamic("x").update(y, z)
```

while

```
dyn.x(y)
```

まだです

```
dyn.applyDynamic("x")(y)
```

これをっておくことがです。そうでないは、かれずにをきして、なエラ―をきこすがあります。

オンラインでびしをむ https://riptutorial.com/ja/scala/topic/8296/びし

44: のメソッドSAMタイプ

メソッドは、 Java 8でされたタイプであり、1つのメンバしかありません。

Examples

ラムダ

これはScala 2.12+および2.11.xバージョンでは`-Xexperimental -Xfuture`コンパイラフラグきでのみできます。

SAMタイプは、ラムダをしてできます。

2.11.8

```
trait Runnable {
  def run(): Unit
}

val t: Runnable = () => println("foo")
```

は、オプションでのメンバをつことができます

2.11.8

```
trait Runnable {
  def run(): Unit
  def concrete: Int = 42
}

val t: Runnable = () => println("foo")
```

オンラインでのメソッドSAMタイプをむ https://riptutorial.com/ja/scala/topic/3664/のメソッド-samタイプ-

45:

Examples

リフレクションをしたクラスのみみ

```
import scala.reflect.runtime.universe._
val mirror = runtimeMirror(getClass.getClassLoader)
val module = mirror.staticModule("org.data.TempClass")
```

オンラインでをむ https://riptutorial.com/ja/scala/topic/5824/

46: された

- `objectToSynchronizeOn.synchronized { /* code to run */}`
- `synchronized {/* code to run, can be suspended with wait */}`

Examples

オブジェクトにする

`synchronized`は、のスレッドがじリソースにアクセスするのをぐのにつレベルののです。 Javaをしたjvmの 。

```
anInstance.synchronized {
  // code to run when the intristic lock on `anInstance` is acquired
  // other thread cannot enter concurrently unless `wait` is called on `anInstance` to suspend
  // other threads can continue of the execution of this thread if they `notify` or
`notifyAll` `anInstance`'s lock
}
```

`object`の、シングルトンインスタンスではなく、オブジェクトのクラスでするがあります。

これににする

```
 /* within a class, def, trait or object, but not a constructor */
 synchronized {
   /* code to run when an intrisctic lock on `this` is acquired */
   /* no other thread can get the this lock unless execution is suspended with
    * `wait` on `this`
    */
 }
```

オンラインでされたをむ https://riptutorial.com/ja/scala/topic/3371/された

47: クラス

の、に Apache Sparkなど でのをけるため、クラス･インスタンスの`Serializable`なをすることがベスト･プラクティスです。

Examples

シンプルタイプクラス

クラスは、1つのパラメータをつなる`trait`です。

```
trait Show[A] {
  def show(a: A): String
}
```

クラスをするわりに、サポートされているごとにクラスののインスタンスがされています。これらのをクラスのコンパニオンオブジェクトにすることで、なインポートをせずにのをうことができます。

```
object Show {
  implicit val intShow: Show[Int] = new Show {
    def show(x: Int): String = x.toString
  }

  implicit val dateShow: Show[java.util.Date] = new Show {
    def show(x: java.util.Date): String = x.getTime.toString
  }

  // ..etc
}
```

にされるパラメータにクラスのインスタンスがあることをしたいは、のパラメータをします。

```
def log[A](a: A)(implicit showInstance: Show[A]): Unit = {
  println(showInstance.show(a))
}
```

コンテキストバインドをすることもできます 。

```
def log[A: Show](a: A): Unit = {
  println(implicitly[Show[A]].show(a))
}
```

のとに、の`log`メソッドをびし`log`。 のはコンパイルにします`Show[A]`のがつからない`A`あなたがにす`log`

```
log(10) // prints: "10"
log(new java.util.Date(1469491668401L) // prints: "1469491668401"
```

```
log(List(1,2,3)) // fails to compile with
                 // could not find implicit value for evidence parameter of type
Show[List[Int]]
```

このでは、 `Show`のクラスをしています。これは、ののインスタンスを`String`にするためにされるのクラスです。すべてのオブジェクトがっているにもかかわらず`toString`メソッドを、それがいるかかはずしもではない`toString`なでされています。 `Show`クラスをすると、 `log`されたものが`String`へのなをつことをすることができ`log`。

クラスの

このでは、のクラスのについてします。

```
trait Show[A] {
  def show: String
}
```

あなたがコントロールするクラスをするにはそしてScalaでかれています、クラスをし、コンパニオンオブジェクトににします。 このから`Person`クラスをして`Show`をするをしましょう

```
class Person(val fullName: String) {
  def this(firstName: String, lastName: String) = this(s"$firstName $lastName")
}
```

このクラスは、 `Person`のコンパニオンオブジェクトににすることで、 `Show`をすることができます

```
object Person {
  implicit val personShow: Show[Person] = new Show {
    def show(p: Person): String = s"Person(${p.fullname})"
  }
}
```

コンパニオンオブジェクトはクラスとじファイルになければならないので、 `class Person`と`object Person`がじファイルにです。

Scalaでしないクラスをするには、 クラスのにすように、クラスをし、クラスのコンパニオンオブジェクトににします。

クラスもクラスもしないは、のようににして`import`します。 シンプル･タイプ･クラスのでの`log`メソッドの

```
object MyShow {
  implicit val personShow: Show[Person] = new Show {
    def show(p: Person): String = s"Person(${p.fullname})"
  }
}

def logPeople(persons: Person*): Unit = {
  import MyShow.personShow
  persons foreach { p => log(p) }
```

```
}
```

にをする

Scalaのクラスのはかなりです。をらす1つのは、いわゆる "クラス"をすることです。これらのクラスは、インポートに/をにラップしてをします。

これをするために、まずなクラスをしましょう

```
// The mathematical definition of "Addable" is "Semigroup"
trait Addable[A] {
  def add(x: A, y: A): A
}
```

に、をしますクラスをインスタンスします。

```
object Instances {

  // Instance for Int
  // Also called evidence object, meaning that this object saw that Int learned how to be
added
  implicit object addableInt extends Addable[Int] {
    def add(x: Int, y: Int): Int = x + y
  }

  // Instance for String
  implicit object addableString extends Addable[String] {
    def add(x: String, y: String): String = x + y
  }

}
```

では、Addableインスタンスをすることはにです。

```
import Instances._
val three = addableInt.add(1,2)
```

むしろ`1.add(2)`くだけです。したがって、たちは`Addable`をするをにラップする "Operation Class" "Ops Class"ともばれるを`Addable`ます。

```
object Ops {
  implicit class AddableOps[A](self: A)(implicit A: Addable[A]) {
    def add(other: A): A = A.add(self, other)
  }
}
```

は、しい`add`を、それが`Int`と`String`であるかのようにできます。

```
object Main {

  import Instances._ // import evidence objects into this scope
  import Ops._       // import the wrappers
```

```
  def main(args: Array[String]): Unit = {

    println(1.add(5))
    println("mag".add("net"))
    // println(1.add(3.141)) // Fails because we didn't create an instance for Double

  }
}
```

"Ops"クラスは、 simulacrumライブラリのマクロによってにされます

```
import simulacrum._

@typeclass trait Addable[A] {
  @op("|+|") def add(x: A, y: A): A
}
```

オンラインでクラスをむ https://riptutorial.com/ja/scala/topic/3835/クラス

48:

Examples

ローカル

Scalaにはながみまれています。このメカニズムは「ローカルタイプ」とばれます。

```
val i = 1 + 2                 // the type of i is Int
val s = "I am a String"       // the type of s is String
def squared(x : Int) = x*x    // the return type of squared is Int
```

コンパイラは、からのをできます。に、メソッドのりは、メソッドからされるとであるため、することができます。のは、のなにします。

```
val i: Int = 1 + 2
val s: String = "I am a String"
def squared(x : Int): Int = x*x
```

タイプとジェネリックス

Scalaコンパイラは、メソッドがびされるとき、またはジェネリッククラスがインスタンスされるときにパラメータをすることもできます

```
case class InferedPair[A, B](a: A, b: B)

val pairFirstInst = InferedPair("Husband", "Wife")  //type is InferedPair[String, String]

// Equivalent, with type explicitly defined
val pairSecondInst: InferedPair[String, String]
                      = InferedPair[String, String]("Husband", "Wife")
```

ののは、Java 7でされたDiamond Operatorにています。

の

Scalaのがしないシナリオがあります。たとえば、コンパイラはメソッドパラメータのをすることはできません。

```
def add(a, b) = a + b  // Does not compile
def add(a: Int, b: Int) = a + b // Compiles
def add(a: Int, b: Int): Int = a + b // Equivalent expression, compiles
```

コンパイラは、メソッドのりのをすることはできません。

```
// Does not compile
```

```
def factorial(n: Int) = if (n == 0 || n == 1) 1 else n * factorial(n - 1)
// Compiles
def factorial(n: Int): Int = if (n == 0 || n == 1) 1 else n * factorial(n - 1)
```

もしないようにする

このブログのにづいています。

のようなメソッドがあるとします。

```
  def get[T]: Option[T] = ???
```

ジェネリックパラメータをせずにびそうとすると、 `Nothing`はされますが、これはのにはあまりにちませんそのはではありません。のソリューションでは、 `NotNothing`コンテキストバインドによって、されるをせずにメソッドをすることをぐことができますこのでは、 `RuntimeClass`も`Nothing`ではなく`ClassTags` `RuntimeClass`がされます。

```
@implicitNotFound("Nothing was inferred")
sealed trait NotNothing[-T]

object NotNothing {
  implicit object notNothing extends NotNothing[Any]
  //We do not want Nothing to be inferred, so make an ambigous implicit
  implicit object `\n The error is because the type parameter was resolved to Nothing` extends
NotNothing[Nothing]
  //For classtags, RuntimeClass can also be inferred, so making that ambigous too
  implicit object `\n The error is because the type parameter was resolved to RuntimeClass`
extends NotNothing[RuntimeClass]
}

object ObjectStore {
  //Using context bounds
  def get[T: NotNothing]: Option[T] = {
    ???
  }

  def newArray[T](length: Int = 10)(implicit ct: ClassTag[T], evNotNothing: NotNothing[T]):
Option[Array[T]] = ???
}
```

```
object X {
  //Fails to compile
  //val nothingInferred = ObjectStore.get

  val anOption = ObjectStore.get[String]
  val optionalArray = ObjectStore.newArray[AnyRef]()

  //Fails to compile
  //val runtimeClassInferred = ObjectStore.newArray()
}
```

オンラインでをむ https://riptutorial.com/ja/scala/topic/4918/

49: の

- {clauses}のために
- {}をむために
- のために
- をむ

パラメーター

パラメータ	
にとって	forループ/をするためになキーワード
	forがするりしとフィルタリング。
	コレクションをしたり、「」したいにします。 `yield`をすると、 `for`りのが`Unit`ではなく、コレクションになります。
	できるforの。

Examples

ベーシック・フォー・ループ

```
for (x <- 1 to 10)
  println("Iteration number " + x)
```

これは、`x`を`1`から`10`までりし、そのでかをうことをしています。の`for`のりのは`Unit`です。

のための

これは、for-loopのフィルタと`yield`をして 'sequence comprehension'をするをしています。

```
for ( x <- 1 to 10 if x % 2 == 0)
  yield x
```

このはのとおりです。

```
scala.collection.immutable.IndexedSeq[Int] = Vector(2, 4, 6, 8, 10)
```

のためのAは、とそのフィルタにづいてしいコレクションをするがあるにです。

ネストされたForループ

これは、のをするをしています。

```
for {
  x <- 1 to 2
  y <- 'a' to 'd'
} println("(" + x + "," + y + ")")
```

なおtoここにりオペレータでを 。をしてくださいここで 。

これによってがされます。

```
(1,a)
(1,b)
(1,c)
(1,d)
(2,a)
(2,b)
(2,c)
(2,d)
```

これは、ブラケットのわりにかっこをしたのであることにしてください。

```
for (
  x <- 1 to 2
  y <- 'a' to 'd'
) println("(" + x + "," + y + ")")
```

すべてのみわせをのベクトルにするyield 、をvalすることができます

```
val a = for {
  x <- 1 to 2
  y <- 'a' to 'd'
} yield "(%s,%s)".format(x, y)
// a: scala.collection.immutable.IndexedSeq[String] = Vector((1,a), (1,b), (1,c), (1,d),
(2,a), (2,b), (2,c), (2,d))
```

のためのモナド

モナドのオブジェクトがあるは、 'for comprehension'をってのみわせをることができます

```
for {
   x <- Option(1)
   y <- Option("b")
   z <- List(3, 4)
} {
    // Now we can use the x, y, z variables
    println(x, y, z)
    x  // the last expression is *not* the output of the block in this case!
}
```

```
// This prints
// (1, "b", 3)
// (1, "b", 4)
```

このブロックのりのは`Unit`です。

オブジェクトがじモナド`M`えば`Option`である、 `yield`をすると`Unit`わりに`M`のオブジェクトがされます。

```
val a = for {
   x <- Option(1)
   y <- Option("b")
} yield {
    // Now we can use the x, y variables
    println(x, y)
    // whatever is at the end of the block is the output
    (7 * x, y)
}

// This prints:
// (1, "b")
// The val `a` is set:
// a: Option[(Int, String)] = Some((7,b))
```

`yield`キーワードは、モナドタイプ`Option`と`List`がしているのではできないことにしてください。そうしようとすると、コンパイルののエラーがします。

Forループをしてコレクションをする

これは、マップのをするをします

```
val map = Map(1 -> "a", 2 -> "b")
for (number <- map) println(number) // prints (1,a), (2,b)
for ((key, value) <- map) println(value) // prints a, b
```

これは、リストのをするをしています

```
val list = List(1,2,3)
for(number <- list) println(number) // prints 1, 2, 3
```

のためにる

Scala `for`の`for`ものはなだけです。これらの`withFilter`は、subjectの`withFilter`、 `foreach`、`flatMap`、および`map`メソッドをしてされます。このため、これらのメソッドがされているだけ`for`の`for`することができます。

A `for`、パターン`pN`、ジェネレータ`gN`および`cN`するののの`for`である。

```
for(p0 <- x0 if g0; p1 <- g1 if c1) { ??? }
```

... withFilterとforeachをしてネストされたびしにデwithFilterします

```
g0.withFilter({ case p0 => c0  case _ => false }).foreach({
  case p0 => g1.withFilter({ case p1 => c1  case _ => false }).foreach({
    case p1 => ???
  })
})
```

、ののfor / yieldがあります。

```
for(p0 <- g0 if c0; p1 <- g1 if c1) yield ???
```

... withFilterとflatMapまたはmapどちらかをしてネストされたびしにデflatMap map

```
g0.withFilter({ case p0 => c0  case _ => false }).flatMap({
  case p0 => g1.withFilter({ case p1 => c1  case _ => false }).map({
    case p1 => ???
  })
})
```

mapはものでされ、 flatMapはすべてのでされます。

のforは、によってされるをするあらゆるタイプにすることができる。これらのメソッドのりは、であるり、はありません。

オンラインでのをむ https://riptutorial.com/ja/scala/topic/669/の

50: の

Examples

の

ScalaではJavaやのほとんどのとはに、 `if`はステートメントではなくです。それにかかわらず、はじです

```
if(x < 1984) {
   println("Good times")
} else if(x == 1984) {
   println("The Orwellian Future begins")
} else {
   println("Poor guy...")
}
```

である`if`のは、ののをにできるということです。

```
val result = if(x > 0) "Greater than 0" else "Less than or equals 0"
\\ result: String = Greater than 0
```

では、 `if`がされ、 `result`がそののにされている`result`がわかります。

`if`のりのは、すべてののスーパータイプです。つまり、このではりのは`String`です。すべてではないので、 `if`のはのようなをす`if`もっていないで`else`ロジックを、りのがあるがある`Any`

```
val result = if(x > 0) "Greater than 0"
// result: Any = Greater than 0
```

されるがない `println`ようなだけがブランチでされるなど、りのは`Unit`

```
val result = if(x > 0) println("Greater than 0")
// result: Unit = ()
```

`if` Scalaでははどのようにていたの。こののために、Scalaにはそのようながありません。これはです。

コンテンツが1の、 `if`でをすることができます。

オンラインでのをむ https://riptutorial.com/ja/scala/topic/4171/の

51:

- ATrait {...}
- クラスAClass...はATrait {...}をします
- クラスAClassはATclassでBClassをします
- クラスAClassがATraitをBTraitで
- クラスAClassは、CTraitでBTraitとATraitをします。
- クラスATraitはBTraitをする

Examples

によるスタックな

をって、クラスのメソッドをスタックなでることができます。

のは、をどのようにみねるかをしています。のはです。なるのをして、なるがされる。

```
class Ball {
  def roll(ball : String) = println("Rolling : " + ball)
}

trait Red extends Ball {
  override def roll(ball : String) = super.roll("Red-" + ball)
}

trait Green extends Ball {
  override def roll(ball : String) = super.roll("Green-" + ball)
}

trait Shiny extends Ball {
  override def roll(ball : String) = super.roll("Shiny-" + ball)
}

object Balls {
  def main(args: Array[String]) {
    val ball1 = new Ball with Shiny with Red
    ball1.roll("Ball-1") // Rolling : Shiny-Red-Ball-1

    val ball2 = new Ball with Green with Shiny
    ball2.roll("Ball-2") // Rolling : Green-Shiny-Ball-2
  }
}
```

`super`は、ので`roll()`メソッドをびすためにされることにしてください。このようにして、スタックなをできます。スタックなの、メソッドのびしはによってされます。

の

これはScalaのもなのバージョンです。

```
trait Identifiable {
  def getIdentifier: String
  def printIndentification(): Unit = println(getIdentifier)
}

case class Puppy(id: String, name: String) extends Identifiable {
  def getIdentifier: String = s"$name has id $id"
}
```

`Identifiable`なにしてスーパークラスはされていないため、デフォルトでは`AnyRef`クラスから`AnyRef`されます。 `Identifiable`には`getIdentifier`のがないため、 `Puppy`クラスはそれをするがあります。しかし、 `Puppy`のを`printIdentification`から`Identifiable`。

REPLでは

```
val p = new Puppy("K9", "Rex")
p.getIdentifier  // res0: String = Rex has id K9
p.printIndentification()  // Rex has id K9
```

ダイヤモンドのをする

ダイヤモンドのやは、JavaインタフエースにたtraitsをってScalaによってされます。はインターフェースよりもがあり、されたメソッドをめることができます。これにより、ののミックスインとのがられます。

Scalaはのクラスからのをサポートしていませんが、ユーザーは1つのクラスでのをできます

```
trait traitA {
  def name = println("This is the 'grandparent' trait.")
}

trait traitB extends traitA {
  override def name = {
    println("B is a child of A.")
    super.name
  }

}

trait traitC extends traitA {
  override def name = {
    println("C is a child of A.")
    super.name
  }
}

object grandChild extends traitB with traitC

grandChild.name
```

ここで、 `grandChild`は`traitB`と`traitC`からしています`traitA`からし`traitA`。 には、にびされるメソッドをするのもされています。

```
C is a child of A.
B is a child of A.
This is the 'grandparent' trait.
```

`super`が`class`または`trait`メソッドをびすためにされるとき、びしをするためにルールがになることにしてください。 `grandChild`のは`grandChild`ようになります。

grandChild - > traitC - > traitB - > traitA - > AnyRef - > Any

---

はのです

```
trait Printer {
  def print(msg : String) = println (msg)
}

trait DelimitWithHyphen extends Printer {
  override def print(msg : String) {
    println("-------------")
    super.print(msg)
  }
}

trait DelimitWithStar extends Printer  {
  override def print(msg : String) {
    println("*************")
    super.print(msg)
  }
}

class CustomPrinter extends Printer with DelimitWithHyphen with DelimitWithStar

object TestPrinter{
  def main(args: Array[String]) {
    new CustomPrinter().print("Hello World!")
  }
}
```

このプログラムはをします

```
*************
-------------
Hello World!
```

`CustomPrinter`はのようになります。

CustomPrinter - > DelimitWithStar - > DelimitWithHyphen - > Printer - > AnyRef - > Any

スタックなの、スカラは、とばれるメソッドびしのをするために、クラスとをのでします。ルールは、 `super()`したメソッドびしをむメソッドでのみされます。これをとしてえてみましょう

```
class Shape {
  def paint (shape: String): Unit = {
    println(shape)
```

```
  }
}

trait Color extends Shape {
  abstract override def paint (shape : String) {
    super.paint(shape + "Color ")
  }
}

trait Blue extends Color {
  abstract override def paint (shape : String) {
    super.paint(shape + "with Blue ")
  }
}

trait Border extends Shape {
  abstract override def paint (shape : String) {
    super.paint(shape + "Border ")
  }
}

trait Dotted extends Border {
  abstract override def paint (shape : String) {
    super.paint(shape + "with Dotted ")
  }
}

class MyShape extends Shape with Dotted with Blue {
  override def paint (shape : String) {
    super.paint(shape)
  }
}
```

はにします 。この、

1. First `Shape`はのようにされます。

   ```
   Shape -> AnyRef -> Any
   ```

2. その、 `Dotted`はされます。

   ```
   Dotted -> Border -> Shape -> AnyRef -> Any
   ```

3. のは`Blue`です。 、 `Blue`のはのようになります。

   ```
   Blue -> Color -> Shape -> AnyRef -> Any
   ```

   `MyShape`の ステップ2 までは、 `Shape -> AnyRef -> Any`がにしているからです。したがって、されます。したがって、 `Blue`はのようになります。

   ```
   Blue -> Color -> Dotted -> Border -> Shape -> AnyRef -> Any
   ```

4. に`Circle`がされ、なのはのようになります。

   - > - >カラー— - > - >ボーダー— - >シェイプ - > AnyRef - >

このののは、 `super`がのクラスまたはでされるときのメソッドのびしをします。からののメソッドの

が、のでびされます。 `new MyShape().paint("Circle ")`がされると、のようにされます。

```
Circle with Blue Color with Dotted Border
```

のについては、 ここをしてください 。

オンラインでをむ https://riptutorial.com/ja/scala/topic/1056/

52:

- valされた1、_ / *された2の* /= valueToBeExtracted
- valueToBeExtracted match {case extractorextractedValue1、_=> ???}
- valtuple1、tuple2、tuple3= tupleWith3Elements
- オブジェクトFoo {def unapplyfooFooオプション[String] = Somefoo.x; }

Examples

タプルエクストラクタ

$_x$と$_y$はタプルからされます

```
val (x, y) = (1337, 42)
// x: Int = 1337
// y: Int = 42
```

をするには、 _ をします。

```
val (_, y: Int) = (1337, 42)
// y: Int = 42
```

エクストラクタをするには

```
val myTuple = (1337, 42)
myTuple._1  // res0: Int = 1337
myTuple._2  // res1: Int = 42
```

タプルは22のさをすることにされたいので、 $._1$スルー$._{22}$するタプルをすると、なくともそのサイズです。

タプルは、リテラルのシンボリックをするためにできます

```
val persons = List("A." -> "Lovelace", "G." -> "Hopper")
val names = List("Lovelace, A.", "Hopper, G.")

assert {
  names ==
    (persons map { name =>
      s"${name._2}, ${name._1}"
    })
}

assert {
  names ==
    (persons map { case (given, surname) =>
      s"$surname, $given"
    })
}
```

ケースクラスエクストラクタ

ケースクラスは、のボイラープレートコードがにまれるクラスです。これのメリットの1つは、Scalaではケースクラスでをいやすくすることです。

```
case class Person(name: String, age: Int)  // Define the case class
val p = Person("Paola", 42)  // Instantiate a value with the case class type

val Person(n, a) = p  // Extract values n and a
// n: String = Paola
// a: Int = 42
```

こので、nびaであるvalプログラムのSをそのようにアクセスすることができるそれらは、Pから「」されているとわれています。ける

```
val p2 = Person("Angela", 1337)

val List(Person(n1, a1), Person(_, a2)) = List(p, p2)
// n1: String = Paola
// a1: Int = 42
// a2: Int = 1337
```

ここでは2つのなことがかります

- は「い」レベルでうことができます。れにされたオブジェクトのプロパティをできます。
- すべてのをするはありません。ワイルドカード_は、そののがでもよいことをし、されます。 valはされません。

に、コレクションのマッチングをにすることができます。

```
val ls = List(p1, p2, p3)  // List of Person objects
ls.map(person => person match {
  case Person(n, a) => println("%s is %d years old".format(n, a))
})
```

ここでは、にチェックするためにをするコードっているpersonいるPerson オブジェクトとすぐにたちがにをきしnと。 a

- カスタム

unapplyメソッドをし、 Optionのをすことで、カスタムをすることができます。

```
class Foo(val x: String)

object Foo {
  def unapply(foo: Foo): Option[String] = Some(foo.x)
}

new Foo("42") match {
  case Foo(x) => x
}
```

```
// "42"
```

されるunapplyは、されるがgetおよびisEmptyメソッドをする、 Optionのものであるがあります。このでは、 Barはこれらのメソッドでされ、 unapplyはBarインスタンスをします。

```
class Bar(val x: String) {
  def get = x
  def isEmpty = false
}

object Bar {
  def unapply(bar: Bar): Bar = bar
}

new Bar("1337") match {
  case Bar(x) => x
}
// "1337"
```

unapplyのりのは、のgetおよびisEmptyをたさないなケースでもあるBooleanにすることもできます。ただし、このでは、 DivisibleByTwoはクラスではなくオブジェクトであり、パラメータをらないしたがって、そのパラメータはバインドできません。

```
object DivisibleByTwo {
  def unapply(num: Int): Boolean = num % 2 == 0
}

4 match {
  case DivisibleByTwo() => "yes"
  case _ => "no"
}
// yes

3 match {
  case DivisibleByTwo() => "yes"
  case _ => "no"
}
// no
```

unapplyは、クラスではなく、クラスのコンパニオンオブジェクトにunapplyれることにunapplyください。のは、このをすればになります。

エクストラクタの

ケースクラスにちょうど2つのがある、そのエクストラクタはインミックスでできます。

```
case class Pair(a: String, b: String)
val p: Pair = Pair("hello", "world")
val x Pair y = p
//x: String = hello
//y: String = world
```

2タプルをすはこのようにします。

```
object Foo {
    def unapply(s: String): Option[(Int, Int)] = Some((s.length, 5))
}
val a Foo b = "hello world!"
//a: Int = 12
//b: Int = 5
```

プログラム

グループされたをつをとしてできます。

```
scala> val address = """(.+):(\d+)""".r
address: scala.util.matching.Regex = (.+):(\d+)

scala> val address(host, port) = "some.domain.org:8080"
host: String = some.domain.org
port: String = 8080
```

していない、に`MatchError`がスローされることにしてください。

```
scala> val address(host, port) = "something not a host and port"
scala.MatchError: something not a host and port (of class java.lang.String)
```

エクストラクタのるいは、からのをするためにできます。これは、がしたにのをできるようにするにちます。

たとえば、 Windowsでできるさまざまなユーザーをとしてえてみましょう。

```
object UserPrincipalName {
  def unapply(str: String): Option[(String, String)] = str.split('@') match {
    case Array(u, d) if u.length > 0 && d.length > 0 => Some((u, d))
    case _ => None
  }
}

object DownLevelLogonName {
  def unapply(str: String): Option[(String, String)] = str.split('\\') match {
    case Array(d, u) if u.length > 0 && d.length > 0 => Some((d, u))
    case _ => None
  }
}

def getDomain(str: String): Option[String] = str match {
  case UserPrincipalName(_, domain) => Some(domain)
  case DownLevelLogonName(domain, _) => Some(domain)
  case _ => None
}
```

にはさせることができるタイプをげることによって、のをすエクストラクタをすることがです

```
object UserPrincipalName {
  def unapply(obj: Any): Option[(String, String)] = obj match {
    case upn: UserPrincipalName => Some((upn.username, upn.domain))
```

```
    case str: String => str.split('@') match {
      case Array(u, d) if u.length > 0 && d.length > 0 => Some((u, d))
      case _ => None
    }
    case _ => None
  }
}
```

に、は、 `tryParse`ようなのメソッドにされる`Option`パターンのなです。

```
UserPrincipalName.unapply("user@domain") match {
  case Some((u, d)) => ???
  case None => ???
}
```

オンラインでをむ https://riptutorial.com/ja/scala/topic/930/

53: の

このは、Scala 2.10.0にあります。

Examples

Helloの

sインターポレータは、ののをします。

```
val name = "Brian"
println(s"Hello $name")
```

に "Hello Brian"をコンソールにします。

fインターポレータをしたきの

```
val num = 42d
```

*f*をしてnum2をする

```
println(f"$num%2.2f")
42.00
```

*e*をしてをしてnumをする

```
println(f"$num%e")
4.200000e+01
```

numと16で

```
println(f"$num%a")
0x1.5p5
```

そののについては、のURLをしてください。
https://docs.oracle.com/javase/6/docs/api/java/util/Formatter.html#detail

リテラルでのの

をしてをリテラルにすることができます。

```
def f(x: String) = x + x
val a = "A"

s"${a}"    // "A"
```

```
s"${f(a)}" // "AA"
```

がなければ、scalaは$ このは f のにをするだけです。 fからStringへのなはないので、のではです。

```
s"$f(a)"  // compile-time error (missing argument list for method f)
```

カスタムインターポレータ

みみのものにえてカスタムのインターポレータをすることはです。

```
my"foo${bar}baz"
```

コンパイラによってのようにされます。

```
new scala.StringContext("foo", "baz").my(bar)
```

scala.StringContextはmyメソッドがありません。したがって、のによってできます。みみsインターポレータとじるいのカスタムは、のようにされます。

```
implicit class MyInterpolator(sc: StringContext) {
  def my(subs: Any*): String = {
    val pit = sc.parts.iterator
    val sit = subs.iterator
    // Note parts.length == subs.length + 1
    val sb = new java.lang.StringBuilder(pit.next())
    while(sit.hasNext) {
      sb.append(sit.next().toString)
      sb.append(pit.next())
    }
    sb.toString
  }
}
```

そして、 my"foo${bar}baz"はmy"foo${bar}baz"ようにdesugarでしょう

```
new MyInterpolation(new StringContext("foo", "baz")).my(bar)
```

のやりのにはないことにしてください。これは、のにすように、をにしてのオブジェクトをすることができるいをきます。

```
case class Let(name: Char, value: Int)

implicit class LetInterpolator(sc: StringContext) {
  def let(value: Int): Let = Let(sc.parts(0).charAt(0), value)
}

let"a=${4}" // Let(a, 4)
let"b=${"foo"}" // error: type mismatch
```

```
let"c=" // error: not enough arguments for method let: (value: Int)Let
```

としての

Scalaマクロの[quasiquotes API]でもになっているように、Scalaのをってなパターンマッチャーをすることもです。

`n"p0${i0}p1"`が`new StringContext("p0", "p1").n(i0)`にしていない、 `StringContext`からのなをすることによってがになることは、 `unapply`または`unapplySeq`メソッドをするのプロパティ`n`

として、 `StringContext`からをすることによってパスセグメントをするのプログラムをえてみましょう。その、のいりげを、としてられる[scala.util.matching.Regex]がする`unapplySeq`メソッドにできます 。

```
implicit class PathExtractor(sc: StringContext) {
  object path {
    def unapplySeq(str: String): Option[Seq[String]] =
      sc.parts.map(Regex.quote).mkString("^", "([^/]+)", "$").r.unapplySeq(str)
  }
}

"/documentation/scala/1629/string-interpolation" match {
  case path"/documentation/${topic}/${id}/${_}" => println(s"$topic, $id")
  case _ => ???
}
```

`path`オブジェクトは、のインターポレータとしてもするために`apply`メソッドをすることもできることにしてください。

の

をそのまま し、リテラルをエスケープしないは、 のをすることができます。

```
println(raw"Hello World In English And French\nEnglish:\tHello World\nFrench:\t\tBonjour Le Monde")
```

のをすると、コンソールにのようながされます。

```
Hello World In English And French\nEnglish:\tHello World\nFrench:\t\tBonjour Le Monde
```

のがなければ、 `\n`と`\t`はエスケープされていました。

```
println("Hello World In English And French\nEnglish:\tHello World\nFrench:\t\tBonjour Le Monde")
```

```
Hello World In English And French
English:        Hello World
French:         Bonjour Le Monde
```

オンラインでのをむ https://riptutorial.com/ja/scala/topic/1629/の

54:

- re.findAllInsCharSequenceMatchIterator
- re.findAllMatchInsCharSequence[マッチ]
- re.findFirstInsCharSequenceオプション[]
- re.findFirstMatchInsCharSequenceオプション[Match]
- re.findPrefixMatchInsCharSequenceオプション[Match]
- re.findPrefixOfsCharSequenceオプション[]
- re.replaceAllInsCharSequence、ReplacerMatch => StringString
- re.replaceAllInsCharSequence、replacementStringString
- re.replaceFirstInsCharSequence、replacementStringString
- re.replaceSomeInsCharSequence、ReplacerMatch =>オプション[]
- re.splitsCharSequenceArray []

Examples

の

scala.collection.immutable.StringOpsをしてにされる`r`メソッドは、からscala.util.matching.Regexのインスタンスをします。 Scalaのでまれたは、Javaのとじようにバックスラッシュをエスケープするがないので、ここではです。

```
val r0: Regex = """(\d{4})-(\d${2})-(\d{2})""".r     // :)
val r1: Regex = "(\\d{4})-(\\d{2})-(\\d{2})".r // :(
```

scala.util.matching.Regexは java.util.regex.Patternのラッパーとして ScalaのAPIをしており、サポートされているはじです。つまり、Scalaはのリテラルをサポートしているため、 `x`フラグはによりになり、コメントをにしてパターンをします。

```
val dateRegex = """(?x:
   (\d{4}) # year
  -(\d{2}) # month
  -(\d{2}) # day
)""".r
```

オーバーロードされたバージョンの`r` 、 `def r(names: String*): Regex`パターンマッチにグループをりてることができる`def r(names: String*): Regex`あります。とキャプチャのけがされるため、これはややです。がのでされるにのみしてください。

```
"""(\d{4})-(\d{2})-(\d{2})""".r("y", "m", "d").findFirstMatchIn(str) match {
  case Some(matched) =>
    val y = matched.group("y").toInt
    val m = matched.group("m").toInt
    val d = matched.group("d").toInt
    java.time.LocalDate.of(y, m, d)
  case None => ???
```

```
}
```

のパターンのりし

```
val re = """\((.*?)\)""".r

val str =
"(The)(example)(of)(repeating)(pattern)(in)(a)(single)(string)(I)(had)(some)(trouble)(with)(once)"


re.findAllMatchIn(str).map(_.group(1)).toList
res2: List[String] = List(The, example, of, repeating, pattern, in, a, single, string, I, had,
some, trouble, with, once)
```

オンラインでをむ https://riptutorial.com/ja/scala/topic/2891/

55:

- @AnAnnotation def someMethod = {...}
- @AnAnnotationクラスsomeClass {...}
- @AnnotatioWithArgsannotation_argsdef someMethod = {...}

パラメーター

パラメータ	
@	のトークンがであることをします。
SomeAnnotation	アノテーションの
constructor_args	オプションアノテーションにされる。しない、かっこはです。

Scala-langは、 アノテーションとそのJavaのリストをします。

Examples

の

このサンプルは、のメソッドが`deprecated`ことをします。

```
@deprecated
def anUnusedLegacyMethod(someArg: Any) =  {
  ...
}
```

これは、にのようにすることもできます。

```
@deprecated def anUnusedLegacyMethod(someArg: Any) =  {
  ...
}
```

メインのコンストラクタにをける

```
/**
 * @param num Numerator
 * @param denom Denominator
 * @throws ArithmeticException in case `denom` is `0`
 */
class Division @throws[ArithmeticException](/*no annotation parameters*/) protected (num: Int,
denom: Int) {
    private[this] val wrongValue = num / denom
```

```
    /** Integer number
     *  @param num Value */
    protected[Division] def this(num: Int) {
      this(num, 1)
    }
}
object Division {
  def apply(num: Int) = new Division(num)
  def apply(num: Int, denom: Int) = new Division(num, denom)
}
```

このは`protected`は、じのアノテーションのにるがあります。アノテーションがオプションのパラメータをけれるこの、`@throws`はオプションのをけれます`()`、コンストラクタパラメータのにアノテーションののパラメータリストをするがあります。

のをすることもできます りし 。

ファクトリメソッドのないケースクラスアノテーションにされたもです。

```
case class Division @throws[ArithmeticException]("denom is 0") (num: Int, denom: Int) {
    private[this] val wrongValue = num / denom
}
```

のをする

scala.annotation.StaticAnnotationまたはscala.annotation.ClassfileAnnotationからしたクラスをして、のScalaアノテーションをできます

```
package animals
// Create Annotation `Mammal`
class Mammal(indigenous:String) extends scala.annotation.StaticAnnotation

// Annotate class Platypus as a `Mammal`
@Mammal(indigenous = "North America")
class Platypus{}
```

リフレクションAPIをしてアノテーションをいわせることができます。

```
scala>import scala.reflect.runtime.{universe ⇒ u}

scala>val platypusType = u.typeOf[Platypus]
platypusType: reflect.runtime.universe.Type = animals.reflection.Platypus

scala>val platypusSymbol = platypusType.typeSymbol.asClass
platypusSymbol: reflect.runtime.universe.ClassSymbol = class Platypus

scala>platypusSymbol.annotations
List[reflect.runtime.universe.Annotation] = List(animals.reflection.Mammal("North America"))
```

オンラインでをむ https://riptutorial.com/ja/scala/topic/3783/

56: のオーバーロード

Examples

カスタムの

Scalaの `+`、 `-`、 `*`、 `++`などはなるメソッドです。えば、 `1 + 2`は`1.+(2)`とくことができます。これらののメソッドを「」とびます。

つまり、これらのをして、のでカスタムメソッドをすることができます。

```
class Matrix(rows: Int, cols: Int, val data: Seq[Seq[Int]]){
  def +(that: Matrix) = {
    val newData = for (r <- 0 until rows) yield
      for (c <- 0 until cols) yield this.data(r)(c) + that.data(r)(c)

    new Matrix(rows, cols, newData)
  }
}
```

これらのとしてされたは、のようにできます。

```
val a = new Matrix(2, 2, Seq(Seq(1,2), Seq(3,4)))
val b = new Matrix(2, 2, Seq(Seq(1,2), Seq(3,4)))

// could also be written a.+(b)
val sum = a + b
```

はが1つしかないことにしてください。オペレータのにあるオブジェクトは、そのオペレータののオブジェクトでそれのオペレータとばれます。 1つのをつのScalaメソッドをとしてできます。

> これはとにうべきです。あなたのメソッドが、そのオペレータからされるものをにうにのみ、にはいとみなされます。があるは、 `+`わりに`add`ように、よりなをけてください。

カスタムの

は、のに`unary_`を`unary_`てできます。は、 `unary_+`、 `unary_-`、 `unary_!`されてい`unary_!` `unary_~`

```
class Matrix(rows: Int, cols: Int, val data: Seq[Seq[Int]]){
  def +(that: Matrix) = {
    val newData = for (r <- 0 until rows) yield
      for (c <- 0 until cols) yield this.data(r)(c) + that.data(r)(c)

    new Matrix(rows, cols, newData)
  }

  def unary_- = {
    val newData = for (r <- 0 until rows) yield
```

```
      for (c <- 0 until cols) yield this.data(r)(c) * -1

    new Matrix(rows, cols, newData)
  }
}
```

は、のようにできます。

```
val a = new Matrix(2, 2, Seq(Seq(1,2), Seq(3,4)))
val negA = -a
```

> これはとにうべきです。にされていないをオーバーロードすると、コードのをくがあります。

オンラインでのオーバーロードをむ https://riptutorial.com/ja/scala/topic/2271/のオーバーロード

57:

き

Scalaのスコープは、 `def` 、 `val` 、 `var`または`class` がどこからアクセスできるかをします。

-
-
- プライベート[this]
- プライベート[fromWhere]
- された
- protected [fromWhere]

Examples

パブリックデフォルトスコープ

デフォルトでは、スコープは`public`です。はどこからでもアクセスできます。

```
package com.example {
  class FooClass {
    val x = "foo"
  }
}

package an.other.package {
  class BarClass {
    val foo = new com.example.FooClass
    foo.x // <- Accessing a public value from another package
  }
}
```

プライベートスコープ

スコープがプライベートである、のクラスまたはのクラスののインスタンスからのみアクセスできます。

```
package com.example {
  class FooClass {
    private val x = "foo"
    def aFoo(otherFoo: FooClass) {
      otherFoo.x // <- Accessing from another instance of the same class
    }
  }
  class BarClass {
    val f = new FooClass
    f.x // <- This will not compile
  }
}
```

プライベートなパッケージのスコープ

プライベートにアクセスできるパッケージをできます。

```
package com.example {
  class FooClass {
    private val x = "foo"
    private[example] val y = "bar"
  }
  class BarClass {
    val f = new FooClass
    f.x // <- Will not compile
    f.y // <- Will compile
  }
}
```

オブジェクトのプライベートスコープ

もなスコープは「オブジェクトプライベート」スコープで、オブジェクトのじインスタンスからそのにアクセスできるようにします。

```
class FooClass {
  private[this] val x = "foo"
  def aFoo(otherFoo: FooClass) = {
    otherFoo.x // <- This will not compile, accessing x outside the object instance
  }
}
```

された

protectedスコープでは、のクラスののサブクラスからにアクセスできます。

```
class FooClass {
  protected val x = "foo"
}
class BarClass extends FooClass {
  val y = x // It is a subclass instance, will compile
}
class ClassB {
  val f = new FooClass
  f.x // <- This will not compile
}
```

パッケージ

package protected scopeは、のパッケージののサブクラスからのみにアクセスできるようにします。

```
package com.example {
  class FooClass {
    protected[example] val x = "foo"
```

```
  }
  class ClassB extends FooClass {
    val y = x // It's in the protected scope, will compile
  }
}
package com {
  class BarClass extends com.example.FooClass {
    val y = x // <- Outside the protected scope, will not compile
  }
}
```

オンラインでをむ https://riptutorial.com/ja/scala/topic/9705/

58:

き

しスタイルは、のれのであり、りのを「」としてにすことをみます。のは、でそのをびしてプログラムのをする。をえるの1つはです。 Scalaライブラリは、に`shift` / `reset`れたプリミティブのでられたをもたらします。

ライブラリ https : //github.com/scala/scala-continuations

- reset {...} //きは、むリセットブロックのわりまできます
- shift {...} //びしのにいて、クロ―ジャにすをします
- A @cpsParam [B、C] // Cのをるためにに A => Bをとする
- @cps [A] // @cpsParamのエイリアス[A、A]
- @suspendable // @cpsParamのエイリアス[Unit、Unit]

`shift` `Int.+` `reset`はプリミティブコントロールフロ―で、 `Int.+`はプリミティブであり、 `Long`はプリミティブです。それらはられたがにほとんどすべてのフロ―をするためににできるというでどちらかよりもです。らはにな "アウトオブザボックス"ではありませんが、ライブラリでなAPIをするににきます。

とモナドもにしています。はにすることができるモナド 、およびそのため、モナドはしている`flatMap`がパラメ―タとしてをります。

Examples

コールバックはコンティニュエーションです

```
// Takes a callback and executes it with the read value
def readFile(path: String)(callback: Try[String] => Unit): Unit = ???

readFile(path) { _.flatMap { file1 =>
  readFile(path2) { _.foreach { file2 =>
    processFiles(file1, file2)
  }}
}}
```

`readFile`へののは、 `readFile`がそれをびしてジョブのにプログラムのをけるという`readFile`で、です。

にコールバックになることをするために、々はライブラリをします。

```
reset { // Reset is a delimiter for continuations.
  for { // Since the callback hell is relegated to continuation library machinery.
        // a for-comprehension can be used
```

```
    file1 <- shift(readFile(path1)) // shift has type (((A => B) => C) => A)
    // We use it as (((Try[String] => Unit) => Unit) => Try[String])
    // It takes all the code that occurs after it is called, up to the end of reset, and
    // makes it into a closure of type (A => B).
    // The reason this works is that shift is actually faking its return type.
    // It only pretends to return A.
    // It actually passes that closure into its function parameter (readFile(path1) here),
    // And that function calls what it thinks is a normal callback with an A.
    // And through compiler magic shift "injects" that A into its own callsite.
    // So if readFile calls its callback with parameter Success("OK"),
    // the shift is replaced with that value and the code is executed until the end of reset,
    // and the return value of that is what the callback in readFile returns.
    // If readFile called its callback twice, then the shift would run this code twice too.
    // Since readFile returns Unit though, the type of the entire reset expression is Unit
    //
    // Think of shift as shifting all the code after it into a closure,
    // and reset as resetting all those shifts and ending the closures.
    file2 <- shift(readFile(path2))
  } processFiles(file1, file2)
}

// After compilation, shift and reset are transformed back into closures
// The for comprehension first desugars to:
reset {
  shift(readFile(path1)).flatMap { file1 => shift(readFile(path2)).foreach { file2 =>
processFiles(file1, file2) } }
}
// And then the callbacks are restored via CPS transformation
readFile(path1) { _.flatMap { file1 => // We see how shift moves the code after it into a
closure
  readFile(path2) { _.foreach { file2 =>
    processFiles(file1, file2)
  }}
}}  // And we see how reset closes all those closures
// And it looks just like the old version!
```

するの

`shift`がりの`reset`ブロックのでびされた、それをって`reset`ブロックにをするをすることができます。 `shift`のタイプはに`(((A => B) => C) => A)`ではなく、には`(((A => B) => C) => (A @cpsParam[B, C]))`。そのは、CPSがなをします。 `reset`なしで`shift`をびすは、りのがそのに「」しています。

`reset`ブロックのでは、 `A @cpsParam[B, C]`のは`A`をつようにえますが、それはちょうどふりをしています。をするためにとされるがをつ`A => B`ので、このタイプはらなければならないすメソッドのコードB。 `C`は「の」りであり、CPSのびしのは`C`です。

さて、このは、のScaladocからったものです

```
val sessions = new HashMap[UUID, Int=>Unit]
def ask(prompt: String): Int @suspendable = // alias for @cpsParam[Unit, Unit]. @cps[Unit] is
also an alias. (@cps[A] = @cpsParam[A,A])
  shift {
    k: (Int => Unit) => {
      println(prompt)
      val id = uuidGen
      sessions += id -> k
```

```
    }
  }

def go(): Unit = reset {
  println("Welcome!")
  val first = ask("Please give me a number") // Uses CPS just like shift
  val second = ask("Please enter another number")
  printf("The sum of your numbers is: %d\n", first + second)
}
```

ここで`ask`はマップにをし、でのコードはその「セッション」をしてクエリのをユーザーにすることができます。このようにして、`go`はにライブラリをすることができますが、そのコードはのコードのようにえます。

オンラインでをむ https://riptutorial.com/ja/scala/topic/8312/

59:

- クラスメーター—val metersDoubleはAnyValをします
- タイプメーター—=ダブル

ユニットにクラスをするか、ユニットのライブラリをすることをおめします。

Examples

タイプエイリアス

```
type Meter = Double
```

このシンプルなアプロ—チは、 `Double`のすべてのタイプがそれとがあるように、ユニットのなをっています。

```
type Second = Double
var length: Meter = 3
val duration: Second = 1
length = duration
length = 0d
```

のすべてがコンパイルされます。この、ユニットはコ—ドののタイプをマ—キングするためにのみできますインテントのみ。

バリュ—クラス

```
case class Meter(meters: Double) extends AnyVal
case class Gram(grams: Double) extends AnyVal
```

クラスは、ユニットをエンコ—ドするためのなをします。

```
var length = Meter(3)
var weight = Gram(4)
//length = weight //type mismatch; found : Gram required: Meter
```

`AnyVal`することにより、JVMレベルでののペナルティはなく、のプリミティブこのは`Double` です。

のユニット `Velocity` aka `MeterPerSecond` をにしたいは、このアプロ—チはではありませんが、そのようなにもできるライブラリがあります

- スクワント
-
- ScalaQuantity

オンライ ンでをむ https://riptutorial.com/ja/scala/topic/5966/--

60:

Examples

パ―シャルは、くの、パ―ツのをするためにされます。

```
sealed trait SuperType
case object A extends SuperType
case object B extends SuperType
case object C extends SuperType

val pfA: PartialFunction[SuperType, Int] = {
  case A => 5
}

val pfB: PartialFunction[SuperType, Int] = {
  case B => 10
}

val input: Seq[SuperType] = Seq(A, B, C)

input.map(pfA orElse pfB orElse {
  case _ => 15
}) // Seq(5, 10, 15)
```

このでは、 `orElse`メソッドとののでがされます。、りのすべてのにするがされます。に、これらののみわせは、としてく。

このパターンは、、コードパスのディスパッチャをにさせることができるをするためにされます。これは、たとえば、 Akka Actorのメソッドではです。

`collect`での

は、のなとしてよくされますが、なワイルドカードのマッチ `case _` をめることで、のメソッドではそのがです。なScalaのになの1つは、Scalaコレクションライブラリでされている`collect`メソッドです。ここで、なは、コレクションのをべて、それらを1つのコンパクトなでれるようにマップおよび/またはフィルタリングするをにする。

1

としてされたがあるとします

```
val sqRoot:PartialFunction[Double,Double] = { case n if n > 0 => math.sqrt(n) }
```

`collect`コンビネータでそれをびすことができます

```
List(-1.1,2.2,3.3,0).collect(sqRoot)
```

のようながあります。

```
List(-1.1,2.2,3.3,0).filter(sqRoot.isDefinedAt).map(sqRoot)
```

2

```
sealed trait SuperType // `sealed` modifier allows inheritance within current build-unit only
case class A(value: Int) extends SuperType
case class B(text: String) extends SuperType
case object C extends SuperType

val input: Seq[SuperType] = Seq(A(5), B("hello"), C, A(25), B(""))

input.collect {
  case A(value) if value < 10   => value.toString
  case B(text) if text.nonEmpty => text
} // Seq("5", "hello")
```

のでは、すべきがいくつかあります。

- パターンマッチングのは、するをにしてにめる。 `case`と`case`がしないは、にされます。
- には、するケースのがされています。
- パターンマッチングは、ガードステートメント `if`とのをバインドします。

な

Scalaには、 のをするpartialというなのがあります。つまり、 `Function1`がされるであれば、`PartialFunction`インスタンスをできます。 パターンマッチングでもされる`case`をして、をですることができます。

```
val pf: PartialFunction[Boolean, Int] = {
  case true => 7
}

pf.isDefinedAt(true) // returns true
pf(true) // returns 7

pf.isDefinedAt(false) // returns false
pf(false) // throws scala.MatchError: false (of class java.lang.Boolean)
```

このにられるように、は、その1パラメータのドメインにわたってされるはない。の`Function1`インスタンスはであるとみなされます。つまり、なすべてのにしてされています。

としての

なは、なScalaではにです。らはしばしばらののためにされている`case`にわたり、をするためにベースの

```
sealed trait SuperType // `sealed` modifier allows inheritance within current build-unit only
case object A extends SuperType
```

```
case object B extends SuperType
case object C extends SuperType

val input: Seq[SuperType] = Seq(A, B, C)

input.map {
  case A => 5
  case _ => 10
} // Seq(5, 10, 10)
```

これにより、のでの`match`のがされます。

```
input.map { item =>
  item match {
    case A => 5
    case _ => 10
  }
} // Seq(5, 10, 10)
```

また、タプルやケースクラスがにされたときに、パターンマッチングをしてパラメータをするためににされます。

```
val input = Seq("A" -> 1, "B" -> 2, "C" -> 3)

input.map { case (a, i) =>
   a + i.toString
} // Seq("A1", "B2", "C3")
```

マップでタプルをするための

これらの3つのマップはであるため、チームがもしやすいとわれるバリエーションをしてください。

```
val numberNames = Map(1 -> "One", 2 -> "Two", 3 -> "Three")

// 1. No extraction
numberNames.map(it => s"${it._1} is written ${it._2}" )

// 2. Extraction within a normal function
numberNames.map(it => {
    val (number, name) = it
    s"$number is written $name"
})

// 3. Extraction via a partial function (note the brackets in the parentheses)
numberNames.map({ case (number, name) => s"$number is written $name" })
```

なはすべてのとするがあります。しないは、にがスローされます。

オンラインでをむ https://riptutorial.com/ja/scala/topic/1638/

61:

Scalaにはファーストクラスのがあります。

とメソッドのい

はScalaのメソッドではありません。はであり、そのようにりてられます。、 `def` `def`をしてされたメソッドは、クラス、、またはオブジェクトにしていなければなりません。

- は、コンパイルに `Function1`などをするクラスにコンパイルされ、ににインスタンスされます。、メソッドは、そのクラス、、またはオブジェクトのメンバーであり、そのにはしません。
- メソッドをにすることはできますが、をメソッドにすることはできません。
- メソッドはパラメータをつことができますが、はパラメータできません。
- メソッドはパラメータのデフォルトをつことができますが、はできません。

Examples

は、されていてをりてられていないです。

は、2つのをり、そのをすです。

```
(x: Int, y: Int) => x + y
```

のを`val`することができます。

```
val sum = (x: Int, y: Int) => x + y
```

は、にののとしてされます。たとえば、コレクションの`map`は、としてのをしています。

```
// Returns Seq("FOO", "BAR", "QUX")
Seq("Foo", "Bar", "Qux").map((x: String) => x.toUpperCase)
```

ののはできますはにされます

```
Seq("Foo", "Bar", "Qux").map((x) => x.toUpperCase)
```

が1つだけの、そののののカッコはできます。

```
Seq("Foo", "Bar", "Qux").map(x => x.toUpperCase)
```

は

のをとしない、さらにいがあります。のスニペットはのようにできます

```
Seq("Foo", "Bar", "Qux").map(_.toUpperCase)
```

_はのをにします。のパラメータをつでは、 _がするたびになるがされます。たとえば、の2つのはです。

```
// Returns "FooBarQux" in both cases
Seq("Foo", "Bar", "Qux").reduce((s1, s2) => s1 + s2)
Seq("Foo", "Bar", "Qux").reduce(_ + _)
```

このをする、_によってされるは、1だけじですることができます。

パラメータなしの

パラメータをしないのをするには、パラメータリストをのままにします。

```
val sayHello = () => println("hello")
```

は、2つのをさせ、のとしてることができる。`f(x)`と`g(x)`えられた`g(x) 、 h(x) = f(g(x))`とにされる。

がコンパイルされると、 `Function1`するにコンパイルされます。 Scalaは、composition `andThen`と`compose`する`Function1`の2つのメソッドをします。 `compose`メソッドは、のにします。

```
val f: B => C = ...
val g: A => B = ...

val h: A => C = f compose g
```

`andThen h(x) = g(f(x))` は、より「DSLのような」をっています

```
val f: A => B = ...
val g: B => C = ...

val h: A => C = f andThen g
```

しいがりてられ、 `f`と`g`じられます。このは、のにしい`h`にされます。

```
def andThen(g: B => C): A => C = new (A => C){
  def apply(x: A) = g(self(x))
}
```

`f`

または`g`いずれかがでする、 `h`をびすと、 `f`と`g`すべてのがそのでします。なのについてもです。

との

```
trait PartialFunction[-A, +B] extends (A => B)
```

すべての`PartialFunction`も`Function1`です。これはなではではありませんが、オブジェクトのにしています。このため、 `Function1`は、に`true` `isDefinedAt`メソッドをするはありません。

でもあるをするには、のをします。

```
{ case i: Int => i + 1 } // or equivalently { case i: Int ⇒ i + 1 }
```

については、 PartialFunctionsをてください 。

オンラインでをむ https://riptutorial.com/ja/scala/topic/477/

62:

Scalaはメソッドとをにのものとしてうためににきにわたります。しかし、フードのでは、らはなです。

メソッドはコードであり、のはありません。

は、 `Function1`ののオブジェクトインスタンスまたはのタイプののです。そのコードは、その`apply`メソッドにまれてい`apply`。には、にすことができるとしてします。

ちなみに、としてのをするは、のサポートをつがする、まさにです 。インスタンスは、このをするためのScalaのアプローチです。

のは、をとしてるか、をすです。しかし、Scalaでは、すべてのがメソッドであるため、のパラメータをけったりすメソッドをえるのがです。したがって、 `Seq`されている`map` 、そのパラメータがであるため「 」とえることができますが、りではありません。です。

Examples

メソッドをとしてする

Scalaコンパイラは、メソッドをにすでにメソッドをにします。

```
object MyObject {
  def mapMethod(input: Int): String = {
    int.toString
  }
}

Seq(1, 2, 3).map(MyObject.mapMethod) // Seq("1", "2", "3")
```

のでは、 `MyObject.mapMethod`はびしではなく、として`map`されます。には、 `map` はそのシグネチャにられるように、 `map`されるがです。 `List[A]` タイプ`A`のオブジェクトのリストの`map`のシグネチャはのとおりです。

```
def map[B](f: (A) ⇒ B): List[B]
```

`f: (A) => B`は、このメソッドびしのパラメータが、 `AA`オブジェクトをり、 `B`のオブジェクトをすであることをします。 `A`と`B`はにされる。のにって、 `mapMethod`は`Int A`をとり、 `String B`するをします。したがって、 `mapMethod`は`map`にすなです。のようにじコードをきすことができます

```
Seq(1, 2, 3).map(x:Int => int.toString)
```

これにより、がインラインされ、なをにすることができます。

パラメータとしての

ではなく、はの3つののいずれかをつことができます。

- 1つのパラメータはであり、あるをします。
- をしますが、そのパラメータのどれもではありません。
- の1つのパラメータはであり、をします。

```
object HOF {
    def main(args: Array[String]) {
    val list =
List(("Srini","E"),("Subash","R"),("Ranjith","RK"),("Vicky","s"),("Sudhar","s"))
    //HOF
     val fullNameList= list.map(n => getFullName(n._1, n._2))

     }

    def getFullName(firstName: String, lastName: String): String = firstName + "." +
lastName
    }
```

ここでmapは、 `getFullName(n._1,n._2)` をパラメータとして `getFullName(n._1,n._2)` ます。 これは **HOF**とばれます。

Scalaは、 `def func(arg: => String)` をしてのをサポートしています。このようなのは、の `String` オブジェクトまたは `String` りのをつをとることがあります。 2のケースでは、のはアクセスでされます。

をてください

```
def calculateData: String = {
  print("Calculating expensive data! ")
  "some expensive data"
}

def dumbMediator(preconditions: Boolean, data: String): Option[String] = {
  print("Applying mediator")
  preconditions match {
    case true => Some(data)
    case false => None
  }
}

def smartMediator(preconditions: Boolean, data: => String): Option[String] = {
  print("Applying mediator")
  preconditions match {
    case true => Some(data)
    case false => None
  }
}

smartMediator(preconditions = false, calculateData)
```

```
dumbMediator(preconditions = false, calculateData)
```

`smartMediator`びしはNoneをし、メッセージ`"Applying mediator"`ます。

`dumbMediator`コールはNoneをし、 `"Calculating expensive data! Applying mediator"`というメッセージを`"Calculating expensive data! Applying mediator"`。

レイジーは、なののオーバーヘッドをするににちます。

オンラインでをむ https://riptutorial.com/ja/scala/topic/1642/

クレジット

S. No		Contributors
1	Scalaをいめる	4444, Andy Hayden, Ani Menon, Community, David G., David Portabella, dk14, Donald.McLean, Gabriele Petronella, Grzegorz Oledzki, implicitdef, isaias-b, J Atkin, Jean, Jonathan, mammothbane, marcospereira, Marek Skiba, mdarwin, Nathaniel Ford, NeoWelkin, Nicofisi, Priya, rolve, Shoe, sschaef, Thomas Andrews, Tyler James Harden, Ven, Vogon Jeltz
2	Gradleをってする	Bianca Tesila, Nathaniel Ford, Rjk
3	Javaの	Andrzej Jozwik, Dan Hulme, Gábor Bakos, mvn, the21st, thekingofkings
4	JSON	ipoteka, John, Muki, Nathaniel Ford, pedrorijo91, suj1th, void, Wogan, zoitol
5	Quasiquotes	gregghz
6	Scala.js	Camilo Sampedro
7	ScalaCheckによるテスト	Andrzej Jozwik
8	ScalaTestによるテスト	Nadim Bahadoor, Nathaniel Ford
9	Scalaの	Gábor Bakos, Shaido, Suminda Sirinath S. Dharmasena
10	Scalaの	Hristo Iliev, Matas Vaitkevicius, Nathaniel Ford, Rjk
11	Whileループ	J Cracknell, Nathaniel Ford
12	XML	Nathaniel Ford, Rockie Yang, vsnyc
13	インプリシット	Andy Hayden, dimitrisli, Gábor Bakos, HTNW, implicitdef, ipoteka, Jose Antonio Jimenez Saez, Michael Zajac, Nathaniel Ford, nattyddubbs, Simon, spiffman, Suma, Timo, vsminkov
14	ヴァール、ヴァル、デフ	Aamir, John Starich, jwvh, linkhyrule5, Nathaniel Ford, Shastick, Shuklaswag, stefanobaghino, ZbyszekKr
15	エラー	Andy Hayden, Graham, John Starich, made raka teja, mnoronha, Nathaniel Ford, Simon, Suma, tacos_tacos_tacos,

		Tzach Zohar
16	オプションクラス	Bruce Lowe, CPS, earldouglas, evan.oman, Governa, John Starich, Matthew Scharley, Nathaniel Ford, R Pieters, ScientiaEtVeritas, suj1th, Tzach Zohar, Vasiliy Levykin
17	カッシング	Adamos Loizou, alphaloop, Amr Gawish, dimitrisli, Luka Jacobowitz, Nathaniel Ford, rjsvaljean, Suma, vise890
18	クラスとオブジェクト	Aamir, Gábor Bakos, mdarwin, mirosval, MSmedberg, Nathaniel Ford, ScientiaEtVeritas, steve, Sudhir Singh, Tzach Zohar, vivek
19	ケースクラス	Andy Hayden, Dan Simon, dk14, Gábor Bakos, HTNW, J Cracknell, keegan, made raka teja, Marc Grue, Nathaniel Ford, pedrorijo91, Rumoku, ScientiaEtVeritas, suj1th, Suma
20	コレクション	Anton, Camilo Sampedro, deepkimo, Donald.McLean, doub1ejack, EdgeCaseBerg, Filippo Vitale, George, implicitdef, ipoteka, Jason, John Starich, Mr D, Nathaniel Ford, raam86, Shastick, Suma, Tundebabzy, Vasiliy Levykin
21	シンボルリテラル	ZbyszekKr
22	スカラズ	chengpohi
23	スカラドック	Camilo Sampedro, Gábor Bakos, Nathaniel Ford
24	ストリーム	jwvh, Nathaniel Ford, Oleg Pyzhcov
25	セルフタイプ	Gábor Bakos, irundaia
26	タイプパラメータ Generics	akauppi, Andy Hayden, Eero Helenius, Nathaniel Ford, vivek
27	タイプレベルプログラミング	J Cracknell
28	タイプ	acjay, J Cracknell, Reactormonk
29	タプル	corvus_192, evan.oman, Lawsy, Nathaniel Ford
30	パーサー	Nathaniel Ford
31	ハイブのユーザー	Camilo Sampedro
32	パターンマッチング	Ali Dehghani, Andrzej Jozwik, Andy Hayden, CPS, Dan Simon, Daniel Werner, Filippo Vitale, Gábor Bakos, implicitdef, insan-e, jilen, jozic, JRomero, Justin Bailey, Louis F., mammothbane, Matt, Nadim Bahadoor, Nathaniel Ford, Peter Neyens, Sergio,

		Shastick, Shoe, Simon, Suma, T.Grottker, user6062072, vdebergue, vsminkov, Yagüe
33	パッケージ	Alex Javarotti, Nathaniel Ford, NetanelRabinowitz
34	ベストプラクティス	corvus_192, ipoteka, Nathaniel Ford, RamenChef, Sarvesh Kumar Singh, Shuklaswag
35	マクロ	gregghz, HTNW, Nathaniel Ford
36	モナド	ipoteka, Nathaniel Ford
37	のスタイルでデータをう	Filippo Vitale
38	コレクション	Nathaniel Ford, Shuklaswag
39		Hoang Ong
40		isaias-b, kevin628, Nathaniel Ford, nukie, Shastick
41		Dmitry Bystritsky, Gábor Bakos, jilen, jwvh, michael_s, ScientiaEtVeritas, teldosas
42		Andy Hayden, Cortwave, Daniel Schröter, Gábor Bakos, implicitdef, ipoteka, Nathaniel Ford, phantomastray, Red Mercury
43	びし	HTNW
44	のメソッドSAMタイプ	Gábor Bakos, Gabriele Petronella, Nathaniel Ford
45		Sachin Janani
46	された	Gábor Bakos
47	クラス	Arseniy Zhizhelev, Daniel C. Sobral, Gábor Bakos, gregghz, Nathaniel Ford, TomTom, Yawar
48		Gábor Bakos, Nathaniel Ford, suj1th
49	の	Andy Hayden, J Cracknell, jwvh, LivingRobot, Nathaniel Ford, ScientiaEtVeritas
50		André Laszlo, Andy Hayden, Donald.McLean, Louis F., Nathaniel Ford, Rumoku, Sudhir Singh, Vogon Jeltz
51		Andy Hayden, Dan Hulme, Dan Simon, Gábor Bakos, gilad hoch, Idloj, J Cracknell, jwvh, knutwalker, Łukasz, Martin Seeler,

		Michael Ahlers, Nathaniel Ford, Suma, W.P. McNeill
52	の	Andy Hayden, Ayberk, Brian, implicitdef, J Cracknell, Nadim Bahadoor
53		dmitry, J Cracknell, Nathaniel Ford
54		Gábor Bakos, Nathaniel Ford, Thomas Matecki
55	のオーバーロード	corvus_192, implicitdef, inzi, mnoronha, Nathaniel Ford, Simon
56		Camilo Sampedro
57		dmitry, HTNW
58		Gábor Bakos
59		acjay, Akash Sethi, David Leppik, dimitrisli, jwvh, Suma, Tzach Zohar
60		Aravindh S, Archeg, Camilo Sampedro, ches, corvus_192, Dawny33, Gábor Bakos, Gabriele Petronella, implicitdef, ipoteka , Jean, jwvh, michael_s, Nathaniel Ford, raam86, rjsvaljean, ScientiaEtVeritas, Shastick, stefanobaghino, Sven Koschnicke, vise890, wheaties
61		acjay, ches, Nathaniel Ford, nukie, Rajat Jain, Srini