はじめに

本資料は、ドワンゴオリジナルの、新卒エンジニア向けの研修資料です。
本資料を用いた講義を受講することで

- プログラミング言語 Scala を用いたアプリケーションを開発できるようになること
- 『Scala スケーラブルプログラミング』（第二版）（通称コップ本）を通読して理解できるようになること
- 『Scala 関数型デザイン&プログラミング—Scalaz コントリビューターによる関数型徹底ガイド』（通称 FP in Scala）を通読して理解できるようになること

が主な目的です。

『Scala スケーラブルプログラミング』は、Scala の言語設計者である Odersky さんらにより書かれた解説書で、Scala の言語機能について詳細に書かれており、Scala プログラマにとってはバイブルと言える本です。第 2 版の内容は Scala 2.8 が対象になっているものの、ほとんどの記述は現在でも役に立つものです[*1]。この本は決して読みにくい本ではないのですが、本の分量が多いのと、関数型など他の言語でのプログラミング経験がないとわかりにくい箇所があります。本テキストを通じて Scala に入門することによって、この『コップ本』も読みやすくなるでしょう。

『Scala 関数型デザイン&プログラミング』は、Scalaz[*2]コントリビューターである Paul Chiusano さん、Rúnar Bjarnason さんらにより書かれた関数型プログラミングについての本です。あくまで関数型プログラミングの本なので Scala や Scalaz についての解説はあまりありません。ただし Scala についての最低限の文法の説明は随時行われています。「本文中で一から自作」というスタンスが徹底されており、型クラスのみならず、List、Stream、Future など、Scala 標準ライブラリに存在するものまで、一から自作しています。そのため「単にパターンを覚えて使う」のではなく、それぞれの関数型の色々な型クラスやパターンが「なぜそういう仕組になっているのか？」という根本的な考え方から説明しています。細かい Scala 自体の言語仕様やテクニック、（標準ライブラリと外部のライブラリ含めた）既存の実在するライブラリの使い方は一切説明せず、とにかく「考え方や概念」のみを重点的に説明しているので、この本を読んで身につけた知識は古くなりにくいという点でおすすめできる

[*1] 原著では Scala 2.12 に対応した第 3 版が出版されています。http://www.artima.com/shop/programming_in_scala_3ed

[*2] 関数型プログラミングを行うための純粋なデータ構造や、ファンクターやモナドといった型クラスとそのインスタンスを提供するライブラリのこと。

一冊です。

読者層としては、

- 大学の情報学部卒である
- Java や C など、1 つ以上のプログラミング言語の講習を受けている
- 何か意味のあるアプリケーションを作ったことがある
- 趣味で Twitter API などを触ったり、プログラミングを行っている

人（相当）を仮定しています。なお、上記の指標はこの資料を読むにあたってのあくまで目安であり、特に大学の情報学部卒でなければ理解できないといったことはありません。ただし、本資料は 1 つ以上のプログラミング言語でアプリケーションを作れることを最低限の前提にしていますので、その点留意ください。

★マークについて

本テキストでは、節や項ごとに、どの程度重要かを示すために★マークを付けていることがあります。以下に基準を示しますので、参考にしてください。

- ★★★： 必ず読むべき（must）
- ★★：読んだ方がいい（should）
- ★：読むといいかも（may）

ライセンス

目次

第 1 章

Scala とは（★）

Scala は 2003 年にスイス連邦工科大学ローザンヌ校（EPFL）の Martin Odersky 教授によって開発されたプログラミング言語です。Scala ではオブジェクト指向と関数型プログラミングの両方を行えるところに特徴があります。また、処理系は JVM 上で動作するため、Java 言語のライブラリのほとんどをシームレスに利用することができます。ただし、Scala はただの better Java ではないので、Scala で効率的にプログラミングするためには Scala の作法を知る必要があります。この文書がその一助になれば幸いです。

なぜ開発言語として Scala を選ぶのか（★）

なぜ Scala を開発言語として選択するのでしょうか。ここでは Scala の優れた点を見ていきたいと思います。

オブジェクト指向プログラミングと関数型プログラミングの統合

Scala という言語の基本的なコンセプトはオブジェクト指向と関数型の統合ですが、それは Java をベースとしたオブジェクト指向言語の上に、関数型の機能を表現することで実現しています。

関数型プログラミングの第一の特徴は、関数を引数に渡したり、返り値にできたりする点ですが[*1]、Scala の世界では関数はオブジェクトです。一見メソッドを引数に渡しているように見えるところでも、そのメソッドを元に関数オブジェクトが生成されて渡されています。もちろんオブジェクト指向の世界でオブジェクトが第一級であることは自然なことです。Scala では、関数をオブジェクトやメソッドと別の概念として導入するのではなく、関数を表現するオブジェクトとして導入することで「統合」しているわけです。

他にも、たとえば Scala の「case class」は、オブジェクト指向の視点では Value Object パターンに近いものと考えられますが、関数型の視点では代数的データ型として考えることができます。また「implicit parameter」はオブジェクト指向の視点では暗黙的に型で解決される引数に見えますが、関

[*1] この特徴を関数が「第一級（first-class）」であると言います。

数型の視点では Haskell の型クラスに近いものと見ることができます。

このように Scala という言語はオブジェクト指向言語に関数型の機能を足すのではなく、オブジェクト指向の概念で関数型の機能を解釈し取り込んでいます。それにより必要以上に言語仕様を肥大化させることなく多様な関数型の機能を実現しています。1 つのプログラムをオブジェクト指向の視点と関数型の視点の両方で考えることはプログラミングに柔軟性と広い視野をもたらします。

関数型プログラミング

最近は関数リテラルを記述できるようにするなど関数型プログラミングの機能を取り込んだプログラミング言語が増えてきていますが、その中でも Scala の関数型プログラミングはかなり高機能であると言えるでしょう。

- case class による代数的データ型
- 静的に網羅性がチェックされるパターンマッチ
- implicit parameter による型クラス
- for によるモナド構文
- モナドの型クラスの定義などに不可欠な高カインド型

以上のように Scala では単に関数が第一級であるだけに留まらず、本格的な関数型プログラミングをするための様々な機能があります。

また、Scala はオブジェクトの不変性（immutability）を意識している言語です。変数宣言は変更可能な var と変更不可能な val が分かれており、コレクションライブラリも mutable と immutable でパッケージがわかれています[*2]。case class もデフォルトでは immutable です。

不変性・参照透過性・純粋性は関数型プログラミングにおいて最も重要な概念と言われていますが、近年、並行・並列プログラミングにおける利便性や性能特性の面で、不変性は関数型に限らず注目を集めており、研究も進んでいます。その知見を応用できるのは Scala の大きな利点と言えるでしょう。

オブジェクト指向プログラミング

Scala が優れているのは関数型プログラミングの機能だけではありません。オブジェクト指向プログラミングにおいても様々な進化を遂げています。

- trait による mixin
- 構造的部分型
- 型パラメータの変位（variance）指定
- self type annotation による静的な依存性の注入
- implicit class（conversion）による既存クラスの拡張
- `private[this]` などの、より細やかなアクセス制限

[*2] http://docs.scala-lang.org/ja/overviews/collections/overview.html

- Java のプリミティブ型がラップされて、全ての値がオブジェクトとして扱える

以上のように Scala ではより柔軟なオブジェクト指向プログラミングが可能になっています。Scala のプログラミングでは特に trait を使った mixin によって、プログラムに高いモジュール性と、新しい設計の視点が得られるでしょう。

Java との互換性

Scala は Java との互換性を第一に考えられた言語です。Scala の型やメソッド呼び出しは Java と互換性があり、Java のライブラリは普通に Scala から使うことができます。大量にある既存の Java ライブラリを使うことができるのは大きな利点です。

また Scala のプログラムは基本的に Java と同じようにバイトコードに変換され実行されるので、Java と同等に高速で、性能予測が容易です。コンパイルされた class ファイルを javap コマンドを使ってどのようにコンパイルされたかを確認することもできます。

運用においても JVM 系のノウハウをそのまま使えることが多いです。実績があり、広く使われている JVM で運用できるのは利点になるでしょう。

非同期プログラミング、並行・分散プログラミング

Scala では非同期の計算を表現する Future が標準ライブラリに含まれており、様々なライブラリで使われています。非同期プログラミングにより、スレッド数を超えるようなクライアントの大量同時のアクセスに対応することができます。

また、他のシステムに問い合わせなければならない場合などにも、スレッドを占有することなく他のシステムの返答を待つことができます。ドワンゴのように内部に多数のシステムがあり、外からの大量アクセスが見込まれる場合、Scala の非同期プログラミングのサポートは大きなプラスになります。

また Scala には Akka という並行・分散プログラミングのためのライブラリがあります。Akka はアクターというスレッドより小さい単位の実行コンポーネントがあり、このアクター間の通信により並行・分散プログラミングをおこないます。Akka では大量のアクターを管理することにより、スケーラビリティや耐障害性を実現しています。Akka もドワンゴのプロダクトで使われています。

第 2 章

sbt をインストールする（★★★）

現実の Scala アプリケーションでは、Scala プログラムを手動でコンパイル[*1]することは非常に稀で、標準的なビルドツールである sbt（Simple Build Tool）というツールを用いることになります。ここでは、sbt のインストールについて説明します。

Mac OS の場合

Mac OS の場合、Homebrew を用いて、

```
$ brew install sbt
```

でインストールするのが楽です。

Windows の場合

chocolatey を用いるのが楽です。chocolatey は Windows 用のパッケージマネージャで活発に開発が行われてます。chocolatey のパッケージには sbt のものもあるので、

```
> choco install sbt
```

とすれば Windows に sbt がインストールされます。

Windows/Linux の場合で、シェル環境で sbt と入力するとバイナリのダウンロードが始まればインストールの成功です。sbt がないと言われる場合、環境変数へ sbt への PATH が通っていないだけですので追加しましょう。Windows での環境変数編集ツールとしては、Rapid Environment Editor が非常に便利です。

[*1] ここで言う “手動で” とは、`scalac` コマンドを直接呼び出すという意味です

REPL と sbt

これからしばらく、REPL（Read Eval Print Loop）機能と呼ばれる対話的な機能を用いて Scala プログラムを試していきますが、それは常に `sbt console` コマンドを経由して行います。

sbt console を起動するには、Windows でも Mac でも

```
$ sbt console
```

と入力すれば OK です。成功すれば、

```
[info] Loading global plugins from /Users/.../.sbt/0.13/plugins
[info] Set current project to sandbox (in build file:/Users/.../sandbox/)
[info] Updating {file:/Users/.../sandbox/}sandbox...
[info] Resolving org.fusesource.jansi#jansi;1.4 ...
[info] Done updating.
[info] Starting scala interpreter...
[info]
Welcome to Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_45).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

のように表示されます。sbt console を終了したい場合は、

```
scala> :quit
```

と入力します。なお、sbt console を立ち上げる箇所には仮のディレクトリを掘っておくことをお勧めします。sbt はカレントディレクトリの下に *target* ディレクトリを生成してディレクトリ空間を汚してしまうからです。

ちなみに、このとき起動される Scala の REPL のバージョンは現在使っている sbt のデフォルトのバージョンになってしまうので、こちらが指定したバージョンの Scala で REPL を起動したい場合は、同じディレクトリに *build.sbt* というファイルを作成し、

```
scalaVersion := "2.11.8"
```

としてやると良いです。この_*.sbt が *sbt* のビルド定義ファイルになるのですが、今は *REPL* に慣れてもらう段階なので、この.sbt_ファイルの細かい書き方についての説明は省略します。

sbt のバージョンについて（★）

この “sbt のバージョンについて” は、最新版を正常にインストールできた場合は、読み飛ばしていただいて構いません。

sbt は `sbt --version` とすると version が表示されます[*2]。このテキストでは基本的に sbt 0.13[*3] がインストールされている前提で説明していきます。0.13 系（0.13.6、0.13.7 など）であれば基本的には問題ないはずですが、無用なトラブルを避けるため、もし過去に少し古いバージョンの sbt をインストールしたことがある場合は、できるだけ最新版を入れておいたほうがいいでしょう。また、もし 0.12 系（0.12.4 など）が入っている場合は、色々と動作が異なり不都合が生じるので、0.12 系の場合は必ず 0.13 系の最新版を入れるようにしてください。

[*2] ハイフンは 1 つではなく 2 つなので注意。

[*3] 具体的にはこれを書いている 2016 年 02 月時点の最新版である 0.13.11。

第 3 章

Scala の基本（★★★）

この節では Scala の基本について、REPL を使った対話形式から始めて順を追って説明していきます。ユーザは Mac OS 環境であることを前提に説明していきますが、Mac OS 依存の部分はそれほど多くないので Windows 環境でもほとんどの場合同様に動作します。

Scala のインストール

これまでで sbt をインストールしたはずですので、特に必要ありません。sbt が適当なバージョンの Scala をダウンロードしてくれます。

REPL で Scala を対話的に試してみよう

Scala プログラムは明示的にコンパイルしてから実行することが多いですが、REPL（Read Eval Print Loop）によって対話的に Scala プログラムを実行することができます。ここでは、まず REPL で Scala に触れてみることにしましょう。

先ほどのように、対話シェル（Windows の場合はコマンドプロンプト）から

```
$ sbt console
```

とコマンドを打ってみましょう。

```
Welcome to Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_45).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

のように表示されるはずです。これで Scala の世界に入る準備ができました。なお、$の部分はシェ

ルプロンプトなので入力しなくて良いです[*1]。

Hello, World!

まずは定番の Hello, World!を表示させてみましょう。

```
scala> println("Hello, World!")
Hello, World!
```

無事、Hello, World!が表示されるはずです。次にちょっと入力を変えて、`println()` をはずしてみましょう。さて、どうなるでしょうか。

```
scala> "Hello, World!"
res1: String = Hello, World!
```

`res1: String = Hello, World` と、先ほどとは違う表示がされましたね。これは、`"Hello, World"`という式の型が `String` であり、その値が`"Hello, World"`であることを示しています。これまで説明していませんでしたが、Scala は静的な型を持つ言語で、実行される前に型が合っていることがチェックされます。C や Java などの言語を知っている人にはおなじみですね。

練習問題

様々なリテラルを REPL で出力してみましょう。

- `0xff`
- `1e308`
- `9223372036854775807L`
- `9223372036854775808L`
- `9223372036854775807`
- `92233720368547758O7.7`
- `1.00000000000000000001 == 1`
- `"\u3042"`
- `"\ud842\udf9f"`

どのように表示されるでしょうか。

簡単な計算

次に簡単な足し算を実行してみましょう。

[*1] Windows の場合は、`$`を`>`に読み替えてください。

```
scala> 1 + 2
res1: Int = 3
```

3 が表示され、その型が Int である事がわかりますね。なお、REPL の表示に出てくる resN というのは、REPL に何かの式（値を返すもの）を入力したときに、その式の値に REPL が勝手に名前をつけたものです。この名前は次のように、その後も使うことができます。

```
scala> res1
res2: Int = 3
```

この機能は REPL で少し長いプログラムを入力するときに便利ですので、活用していきましょう。Int 型には他にも +,-,*,/といった演算子が用意してあり、次のようにして使うことができます。

```
scala> 1 + 2
res2: Int = 3

scala> 2 * 2
res3: Int = 4

scala> 4 / 2
res4: Int = 2

scala> 4 % 3
res5: Int = 1
```

浮動小数点数の演算のためにも、ほぼ同じ演算子が用意されています。ただし、浮動小数点数には誤差があるためその点には注意が必要です。見ればわかるように、Double という Int と異なる型が用意されています。dbl.asInstanceOf[Int] のようにキャストして型を変換することができますが、その場合、浮動小数点数の小数の部分が切り捨てられることに注意してください。

```
scala> 1.0 + 2.0
res6: Double = 3.0

scala> 2.2 * 2
res7: Double = 4.4

scala> 4.5 / 2
res8: Double = 2.25

scala> 4 % 3
res9: Int = 1

scala> 4.0 % 2.0
res10: Double = 0.0

scala> 4.0 % 3.0
res11: Double = 1.0
```

練習問題

これまで出てきた、+,-,*,/,% の演算子を使って好きなように数式を打ち込んでみましょう。

- `2147483647 + 1`
- `9223372036854775807L + 1`
- `1e308 + 1`
- `1 + 0.0000000000000000001`
- `1 - 1`
- `1 - 0.1`
- `0.1 - 1`
- `0.1 - 0.1`
- `0.0000000000000000001 - 1`
- `0.1 * 0.1`
- `20 * 0.1`
- `1 / 3`
- `1.0 / 3`
- `1 / 3.0`
- `3.0 / 3.0`
- `1.0 / 10 * 1 / 10`
- `1 / 10 * 1 / 10.0`

どのような値になるでしょうか。また、その型は何になるでしょうか。

変数の基本

ここまでは変数を使わずに REPL に式をそのまま打ち込んでいましたが、長い式を入力するときにそれでは不便ですね。Scala にも C や Java と同様に変数があり、計算結果を格納することができます。注意しなければいけないのは Scala の変数には `val` と `var` の 2 種類があり、前者は一度変数に値を格納したら変更不可能で、後者は C や Java の通常の変数のように変更可能だということです。`val` は Java の final な変数と同じように考えれば良いでしょう。

それでは、変数を使ってみることにします。Scala では基本的に、`var` はあまり使わず `val` のみでプログラミングします。これは **Scala でプログラミングをする上で大変重要なことなので**忘れないようにしてください。

```
scala> val x = 3 * 2
x: Int = 6
```

これは、変数 x を定義し、それに `3 * 2` の計算結果を格納しています。特筆すべきは、C や Java

と違い、x の型を宣言していないということです。3 * 2 の結果が Int なので、そこから x の型も Int に違いないとコンパイラが推論した結果です。Scala のこの機能を型推論と呼びます。定義した変数の型は Int と推論されたので、その後、別の型、たとえば String 型の値を代入しようとしてもエラーになります。変数の型を宣言していなくてもちゃんと型が決まっているということです。

val を使った変数には値を再代入できないため、型推論の効果がわかりにくいので、var を使った変数で実験してみます。var を使った変数宣言の方法は基本的に val と同じです。

```
scala> var x = 3 * 3
x: Int = 9

scala> x = "Hello, World!"
<console>:8: error: type mismatch;
 found   : String("Hello, World!")
 required: Int
       x = "Hello, World!"
           ^

scala> x = 3 * 4
x: Int = 12
```

ポイントは、

- var を使って宣言した場合も型推論がはたらく（3 * 3 → Int）
- var を使った場合、変数の値を変更することができる（9 → 12）

というところです。

なお、ここまでは変数の型を宣言せずに型推論にまかせて来ましたが、もちろん、明示的に変数の型を宣言することができます。変数の型を宣言するには

(val または var)　変数名 : 型名 = 式

のようにします。

```
scala> val x: Int = 3 * 3
x: Int = 9
```

ちゃんと変数の型を宣言できていますね。

練習問題

これまで出てきた変数と演算子を用いて、より複雑な数式を入力してみましょう。

- Q. ¥3,950,000 を年利率 2.3 %の単利で 8 か月間借り入れた場合の利息はいくらか（円未満切り捨て）

A. ¥60,566

- Q. 定価¥1,980,000 の商品を値引きして販売したところ、原価 1.6 %にあたる¥26,400 の損失となった。割引額は定価の何パーセントであったか

A. 18 %
以上 2 つを、実際に変数を使って Scala のコードで解いてみましょう。

第 4 章

sbt でプログラムをコンパイル・実行する（★★）

前節まででは、REPL を使って Scala のプログラムを気軽に実行してみました。この節では Scala のプログラムを sbt でコンパイルして実行する方法を学びましょう。まずは REPL の時と同様に Hello, World!を表示するプログラムを作ってみましょう。その前に、REPL を抜けましょう。REPL を抜けるには、REPL から以下のように入力します[*1]。

```
>:quit
```

Scala 2.10 までは `exit`、Scala 2.11 以降は `sys.exit` で終了することができますが、これらは REPL 専用のコマンドではなく、今のプロセス自体を終了させる汎用的なメソッドなので REPL を終了させる時には使用しないようにしましょう。

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

`object` や `def` といった新しいキーワードが出てきましたね。これらの詳しい意味はあとで説明するので、ここでは、`scalac` でコンパイルするプログラムはこのような形で定義するものと思ってください。`{}`で囲まれている部分は REPL の場合と同じ、`println("Hello, World!")` ですね。これを *HelloWorld.scala* という名前のファイルに保存します。

上記のプログラムは sbt でコンパイルし、実行することができます。ここでは *sandbox* というディレクトリを作成し、そこにプログラムを置くことにしましょう。

```
sandbox
├── HelloWorld.scala
```

[*1] `:quit` ではなく`:q` のみでも終了できます。

```
└── build.sbt
```

以上のようにファイルを置きます。

今回の *build.sbt* には Scala のバージョンと一緒に `scalac` の警告オプションも有効にしてみましょう。

```
// build.sbt
scalaVersion := "2.11.8"

scalacOptions ++= Seq("-deprecation", "-feature", "-unchecked", "-Xlint")
```

この記述を加えることで `scalac` が

- 今後廃止の予定の API を利用している（`-deprecation`）
- 明示的に使用を宣言しないといけない実験的な機能や注意しなければならない機能を利用している（`-feature`）
- 型消去などでパターンマッチが有効に機能しない場合（`-unchecked`）
- その他、望ましい書き方や落とし穴についての情報（`-Xlint`）

などの警告の情報を詳しく出してくれるようになります。コンパイラのメッセージが親切になるので付けるようにしましょう。

さて、このようにファイルを配置したら *sandbox* ディレクトリに入り、sbt を起動します。すると sbt のプロンプトが出て、sbt のコマンドが入力できるようになります。今回は HelloWorld のプログラムを実行するために `run` コマンドを入力してみましょう。

```
> run
[info] Compiling 1 Scala source to ...
[info] Running HelloWorld
Hello, World!
[success] Total time: 1 s, completed 2015/02/09 15:44:44
```

HelloWorld プログラムがコンパイルされ、さらに実行されて `Hello, World!`と表示されました。`run` コマンドでは `main` メソッドを持っているオブジェクトを探して実行してくれます。

また sbt の管理下の Scala プログラムは `console` コマンドで REPL から呼び出せるようになります。*HelloWorld.scala* と同じ場所に *User.scala* というファイルを作ってみましょう

```
// User.scala
class User(val name: String, val age: Int)

object User {
  def printUser(user: User) = println(user.name + " " + user.age)
}
```

この *User.scala* には `User` クラスと `User` オブジェクトがあり、`User` オブジェクトには `User` の情報を表示する `printUser` メソッドがあります（クラスやオブジェクトの詳細についてはこの後の節

で説明します)。

```
sandbox
├── HelloWorld.scala
├── User.scala
└── build.sbt
```

この状態で `sbt console` で REPL を起動すると、REPL で `User` クラスや `User` オブジェクトを利用することができます。

```
scala> val u = new User("dwango", 13)
u: User = User@20daebd4

scala> User.printUser(u)
dwango 13
```

今後の節では様々なサンプルコードが出てきますが、このように sbt を使うと簡単に自分で試してみることができるので、活用してみてください。

第 5 章

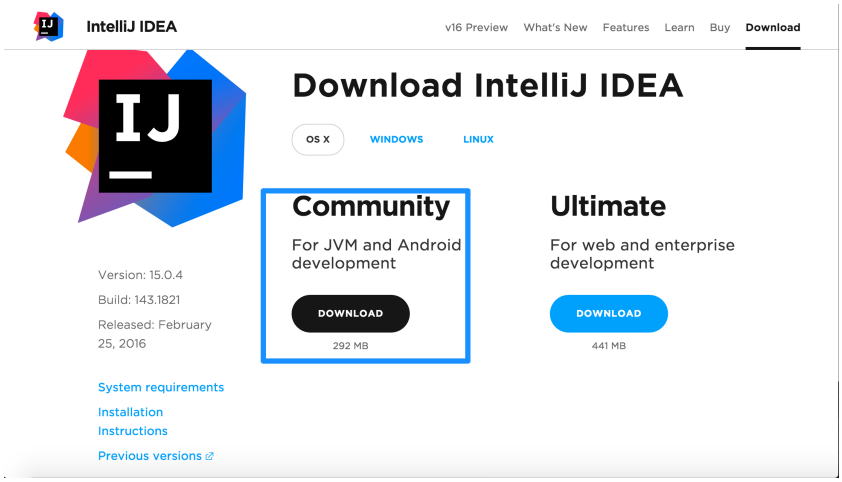
IDE (Integrated Development Environment)

Scala で本格的にコーディングする際は、エディタ（Emacs, Vim, Sublime Text）を使っても構いませんが、IDE を使うとより開発が楽になります。Scala で開発する際の IDE としては IntelliJ IDEA + Scala Plugin と Scala IDE for Eclipse の 2 種類がありますが、IntelliJ IDEA の方が IDE として高機能であり、安定性も高い ため、ここでは IntelliJ IDEA + Scala Plugin のインストール方法と基本的な操作について説明します。

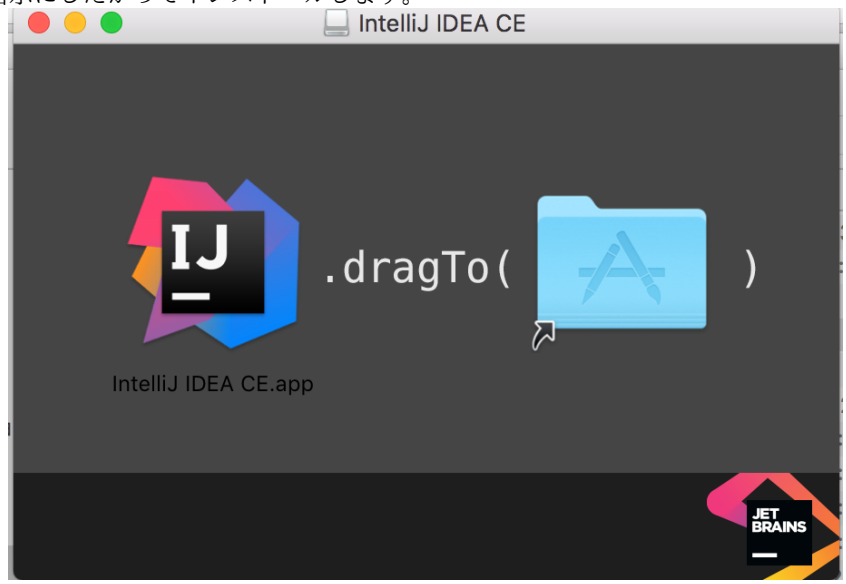
補足：なお、本節で IDE をインストール後も説明は **REPL を使って行いますが**、IDE ベースで学習したい場合は適宜コードを IDE のエディタに貼り付けて実行するなどしてください。

IntelliJ IDEA は JetBrains 社が開発している IDE で、Java 用 IDE として根強い人気を誇っています。有料だが多機能の Ultimate Edition やオープンソース で無料の Community Edition があります。Community Edition でも、Scala プラグインをインストールすることで Scala 開発をすることができるので、本稿では Community Edition をインストール してもらいます。

まず、IntelliJ IDEA の Download ページに移動します。Windows, Mac OS X, Linux の 3 つのタブがあるので、それぞれの OS に応じたタブを選択して、「Download Community」ボタンをクリックしてください。以降、IDEA のスクリーンショットがでてきますが、都合上、Mac OS X 上でのスクリーンショットである ことをご承知ください。

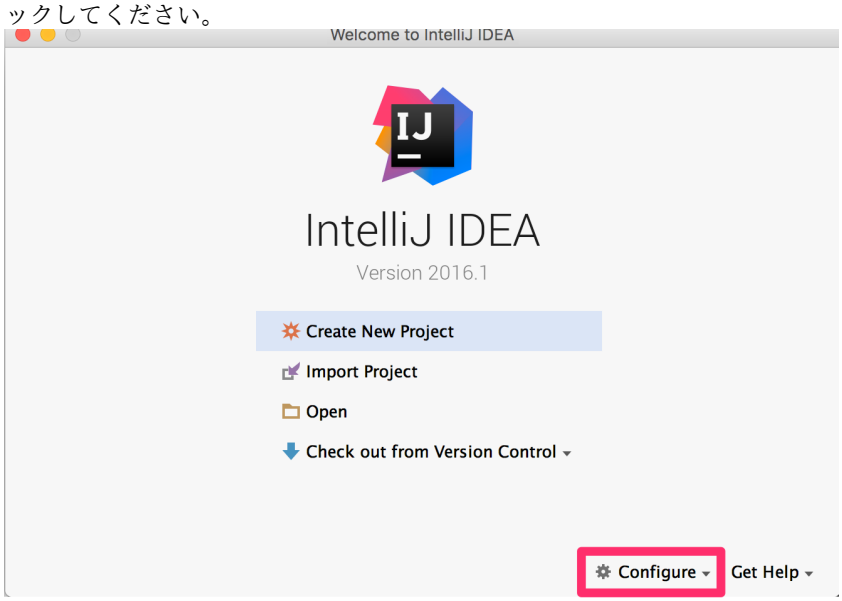


ボタンをクリックすると、ideaIC-${version}.dmg(${version}はその時点での IDEA のバージョン) というファイルのダウンロードが始まるので（Windows の場合、インストーラの exe ファイル)、ダウンロードが終わるのを待ちます。ダウンロードが終了したら、ファイルをダブルクリックして、指示にしたがってインストールします。

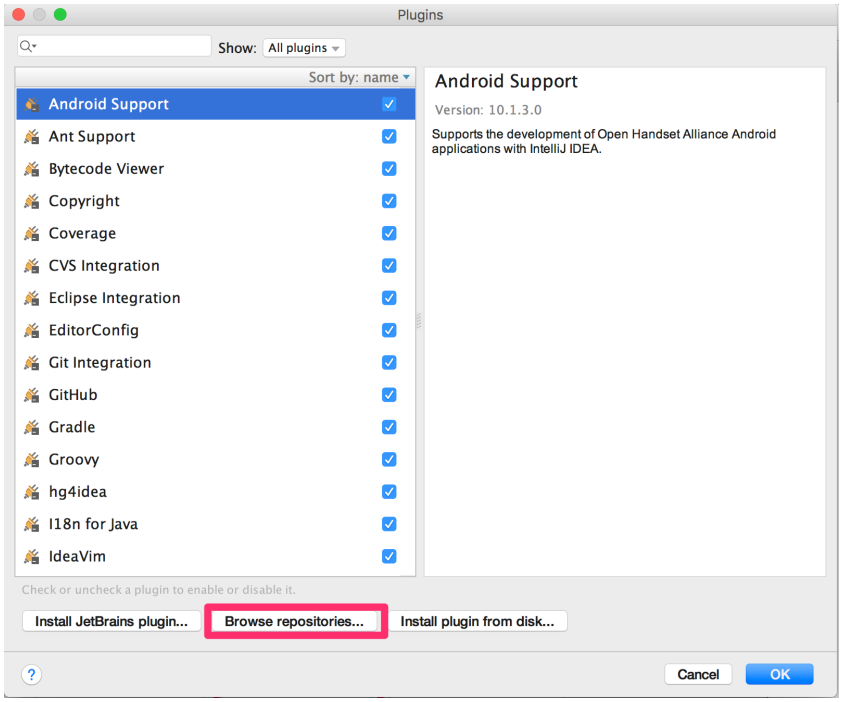


インストールが終わったら起動します。スプラッシュスクリーンが現れてしばらく待つと、

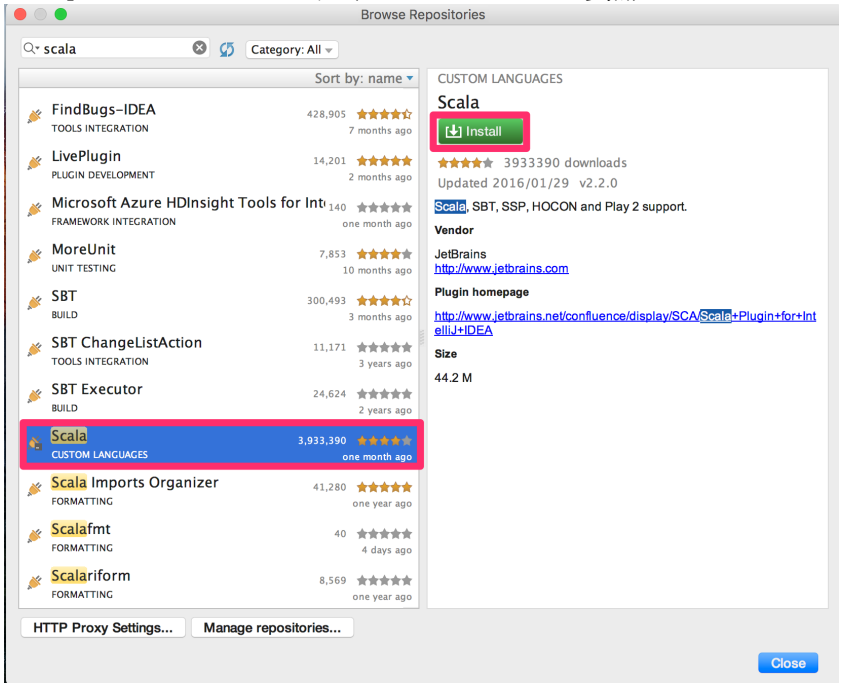
のような画面が表示されるはずです。ここまでで、IDEA のインストールは完了です。次に、IDEA の Scala プラグインをインストールする必要があります。起動画面の `Configure->Plugins` をクリックしてください。



次のような画面が現れるので、その中の「Browse repositories」をクリックします。

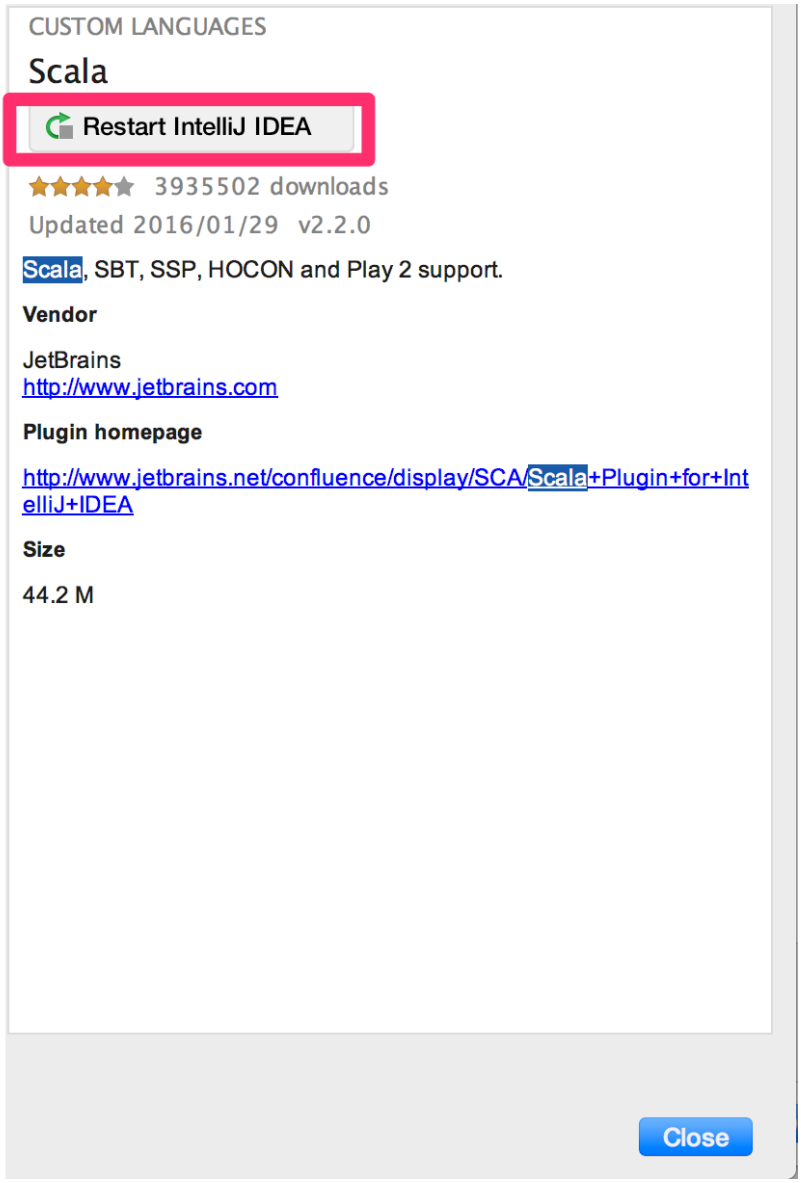


インストール可能なプラグインの一覧が現れるので、検索窓から `scala` で絞り込みをかけて、「Install」でインストールします（スクリーンショット参照）

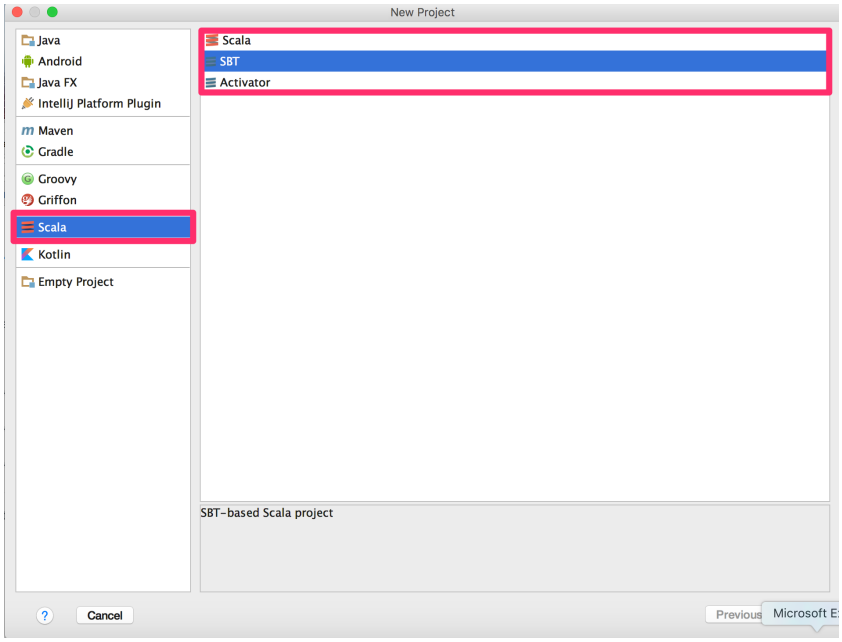


Scala プラグインのダウンロード・インストールには若干時間がかかるのでしばらく待ちます。ダウンロード・インストールが完了したら、次のような画面が現れるので、「Restart IntelliJ IDEA」 を

クリックします。



再起動後、起動画面で「Create New Project」をクリックし、次の画面のようになっていればインストールは成功です。



sbt プロジェクトのインポート

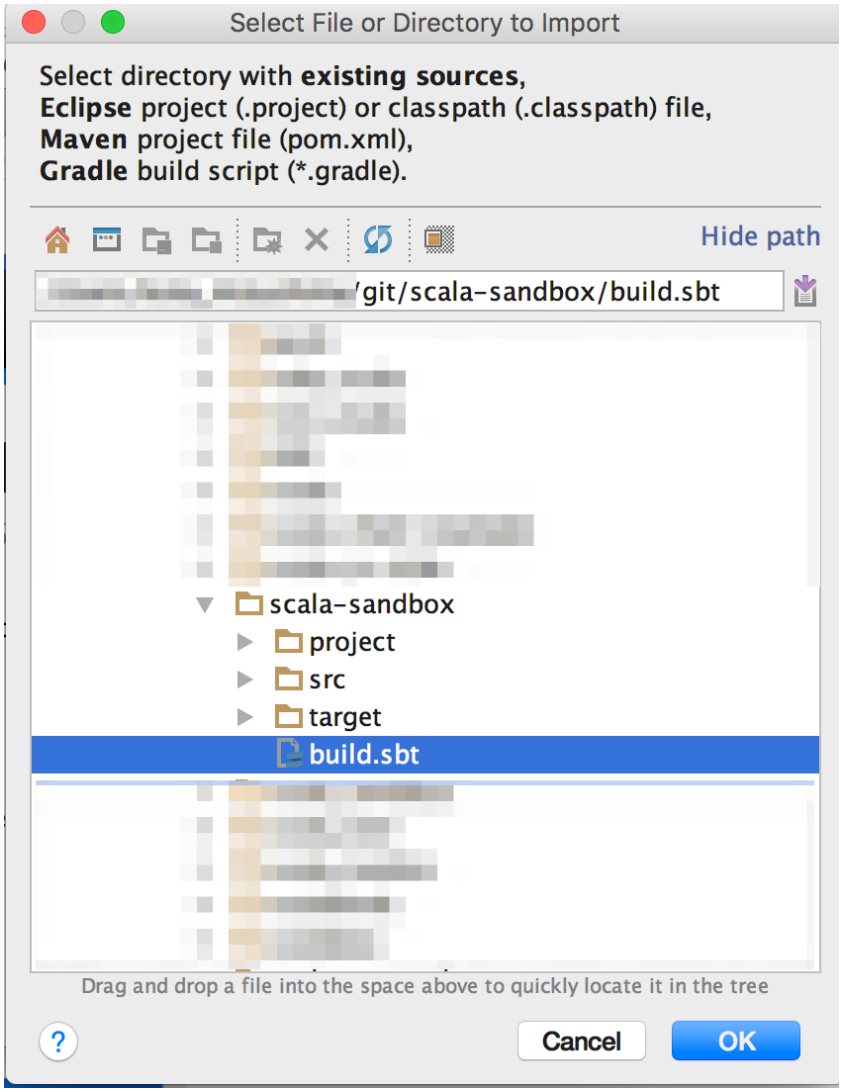
次に、単純な sbt プロジェクトを IntelliJ IDEA にインポートしてみます。今回は、あらかじめ用意しておいた `scala-sandbox` プロジェクトを使います。`scala-sandbox` プロジェクトはここから `git clone` で clone します。`scala-sandbox` プロジェクトは次のような構成になっているはずです。

```
scala-sandbox
├── build.sbt
├── src/main/scala/HelloWorld.scala
├── project/build.properties
└── build.sbt
```

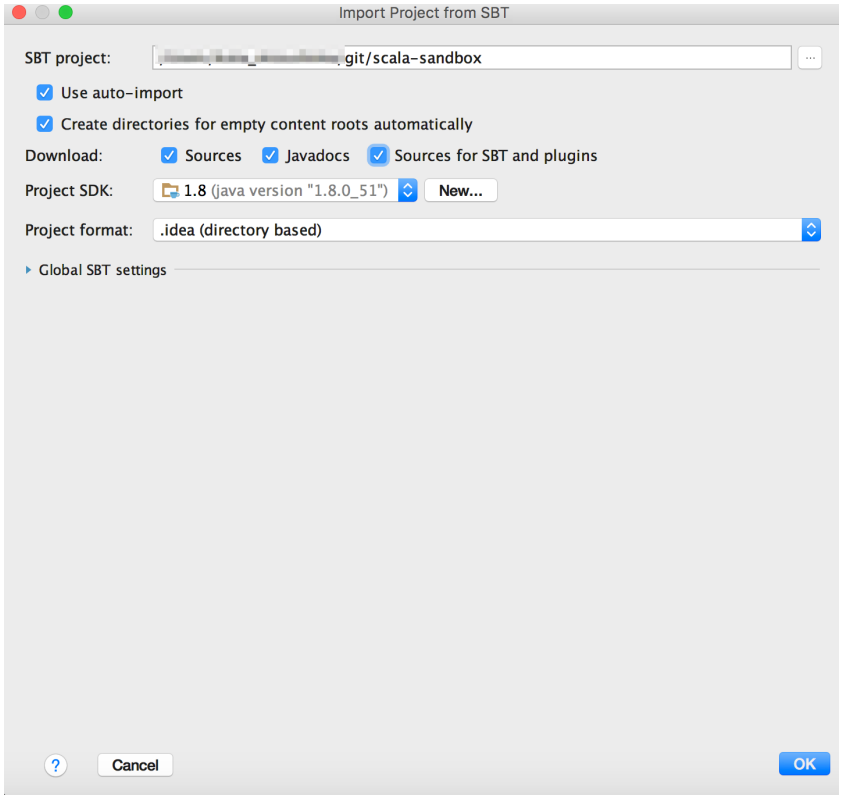
この `scala-sandbox` ディレクトリをプロジェクトとして IntelliJ IDEA に取り込みます。

まず、IntelliJ IDEA の起動画面から、「Import Project」を選択します。

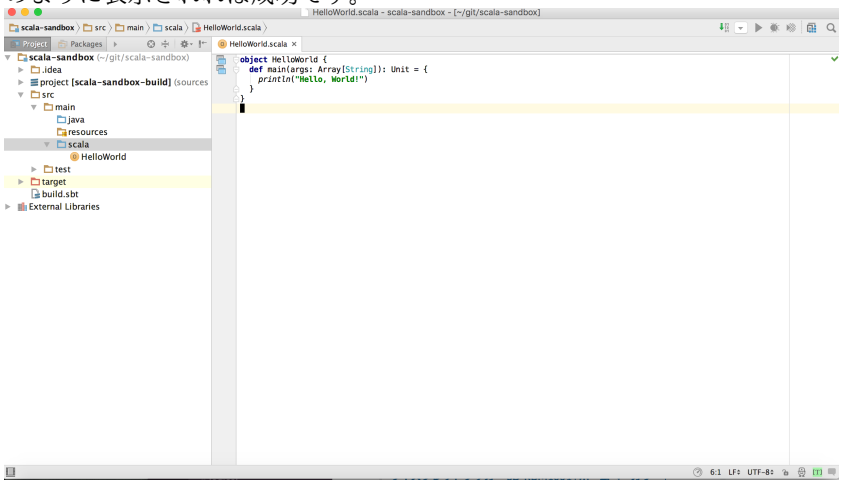
次に、以下のような画面が表示されるので、build.sbt をクリックして、OK します。



すると、さらに次のような画面が表示されるので、赤枠で囲った箇所全てにチェックを入れます。「Project SDK」が空の場合がありますが、その場合は、「New」を選択して自分で、JDK のホームディレクトリにある JDK を指定します。



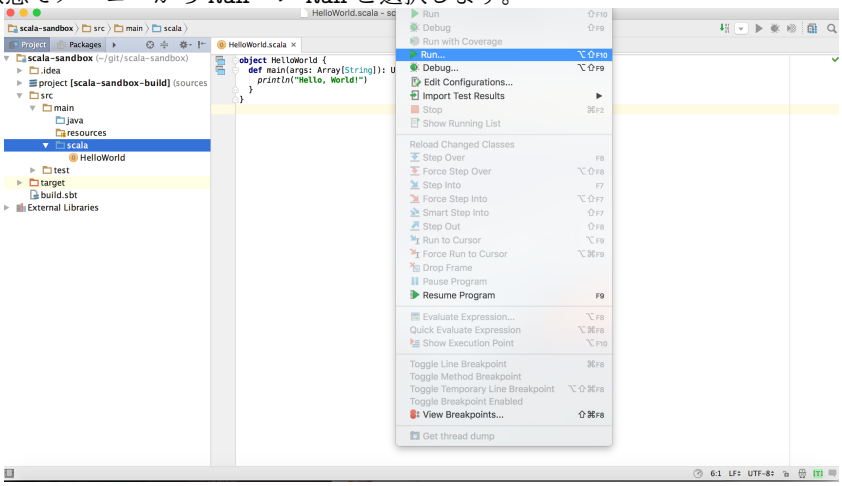
最後に、OK します。最初は、sbt 自体のソースを取得などするため時間がかかりますが、しばらく待つとプロジェクト画面が開きます。そこから、*HelloWorld.scala* を選んでダブルクリックして、以下のように表示されれば成功です。



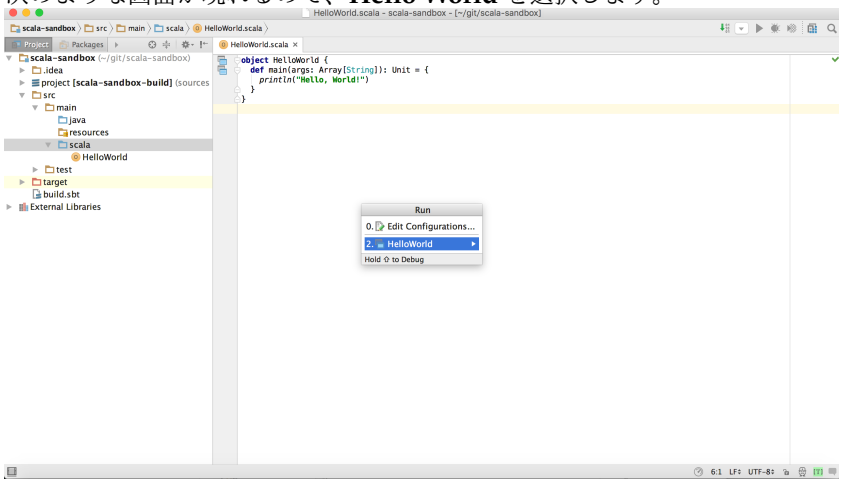
このように、IntelliJ IDEA では、sbt のプロジェクトをインポートして IDE 内で編集することができます。実は、IntelliJ IDEA の以前のバージョンでは sbt-idea という sbt プラグインを使って、IDEA の設定ファイルを自動生成して開くのが普通でした。しかし、この方法は IntelliJ IDEA 14 以降非推奨になったので、本稿では sbt のプロジェクトとしてインポートする方法を取りました。

プログラムを実行する

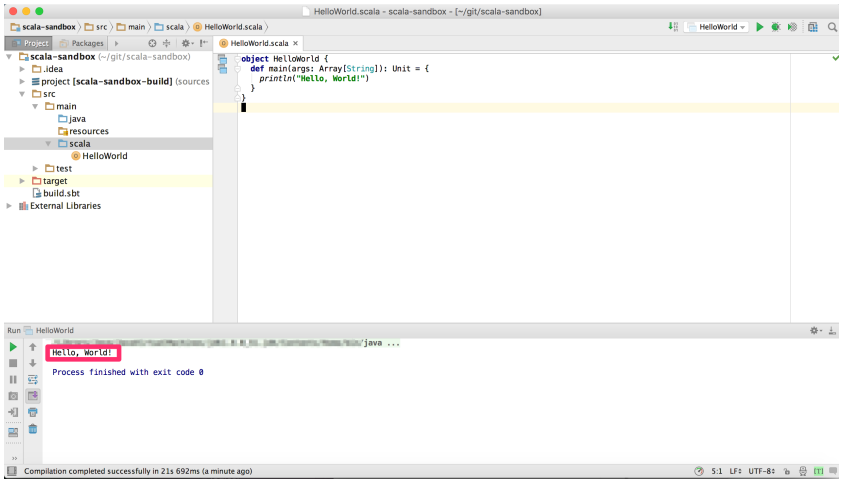
最後に、この HelloWorld プログラムを実行します。*HelloWorld.scala* を IDE のエディタで開いた状態でメニューから `Run -> Run` と選択します。



次のような画面が現れるので、**Hello World** を選択します。



すると、Hello World プログラムの最初のコンパイルが始まるので少し待ちます。その後、次のような画面が表示されたら成功です。



なお、ここでは IntelliJ IDEA のセットアップ方法のみを説明しましたが、最近安定性と機能が向上した Scala IDE for Eclipse を試してみても良いかもしれません。

第 6 章

制御構文

この節では、Scala の制御構文について学びます。通常のプログラミング言語とくらべてそれほど突飛なものが出てくるわけではないので心配は要りません。

「構文」と「式」と「文」という用語について（★）

この節では「構文」と「式」と「文」という用語が入り乱れて使われて少々わかりづらいかもしれないので、先にこの 3 つの用語の解説をしたいと思います。

まず「構文（Syntax）」は、そのプログラミング言語内でプログラムが構造を持つためのルールです。多くの場合、プログラミング言語内で特別扱いされるキーワード、たとえば `class` や `val`、`if` などが含まれ、そして正しいプログラムを構成するためのルールがあります。たとえば `class` の後にはクラス名が続き、クラスの中身は`{`と`}`で括られる、などです。この節は Scala の制御構文を説明するので、処理の流れを制御するようなプログラムを作るためのルールが説明されるわけです。

次に「式（Expression）」は、プログラムを構成する部分のうち、評価することで値になるものです。たとえば `1` や `1 + 2`、`"hoge"`などです。これらは評価することにより、数値や文字列の値になります。

最後に「文（Statement）」ですが、式とは対照的にプログラムを構成する部分のうち、評価しても値にならないものです。たとえば変数の定義である `val i = 1` は評価しても変数 `i` が定義され、`i` の値が `1` になりますが、この定義全体としては値を持ちません。よって、これは文です。

Scala は他の C や Java などの手続き型の言語に比べて、文よりも式になる構文が多いです。Scala では文よりも式を多く利用する構文が採用されています。これにより変数などの状態を出来るだけ排除した分かりやすいコードが書きやすくなっています。

このような言葉の使われ方に注意し、以下の説明を読んでみてください。

{}式（★★★）

`{}`構文の一般形は

```
{ exp1; exp2; ... expN; }
```

となります。exp1 から expN は式です。式が改行で区切られていればセミコロンは省略できます。{}式は exp1 から expN を順番に評価し、expN を評価した値を返します。

次の式では

```
scala> { println("A"); println("B"); 1 + 2 }
A
B
res0: Int = 3
```

A と B が出力され、最後の式である 1 + 2 の結果である 3 が{}式の値になっていることがわかります。

このことは、後ほど記述するメソッド定義などにおいて重要になってきます。Scala では、

```
def foo(): String = {
  "foo" + "foo"
}
```

のような形でメソッド定義をすることが一般的ですが（後述します）、ここで{}は単に{}式であって、メソッド定義の構文に特別に{}が含まれているわけではありません。ただし、クラス定義構文などにおける{}は別です。

if 式（★★★）

if 式は Java の if 文とほとんど同じ使い方をします。if 式の構文は次のようになります。

```
if （条件式） A [else B]
```

条件式は Boolean 型である必要があります。else B は省略することができます。A は条件式が true のときに評価される式で、B は条件式が false のときに評価される式です。

早速 if 式を使ってみましょう。

```
scala> var age = 17
age: Int = 17

scala> if(age < 18) {
     |   "18 歳未満です"
     | } else {
     |   "18 歳以上です"
     | }
res1: String = 18 歳未満です

scala> age = 18
age: Int = 18
```

```
scala> if(age < 18) {
     |   "18 歳未満です"
     | } else {
     |   "18 歳以上です"
     | }
res2: String = 18 歳以上です
```

変更可能な変数 age が 18 より小さいかどうかで別の文字列を返すようにしています。

if 式に限らず、Scala の制御構文は全て式です。つまり必ず何らかの値を返します。Java などの言語で三項演算子?:を見たことがある人もいるかもしれませんが、同じように値が必要な場面で if 式も使えます。

なお、else が省略可能だと書きましたが、その場合は、

```
if（条件式） A else ()
```

と Unit 型の値が補われたのと同じ値が返ってきます。

練習問題

var age: Int = 5 という年齢を定義する変数と var isSchoolStarted: Boolean = false という就学を開始しているかどうかという変数を利用して、1 歳から 6 歳までの就学以前の子どもの場合に “幼児です” と出力し、それ以外の場合は “幼児ではありません” と出力するコードを書いてみましょう。

```
scala> var age: Int = 5
age: Int = 5

scala> var isSchoolStarted: Boolean = false
isSchoolStarted: Boolean = false

scala> if(1 <= age && age <= 6 && !isSchoolStarted) {
     |   println("幼児です")
     | } else {
     |   println("幼児ではありません")
     | }
幼児です
```

while 式（★★）

while 式の構文は Java のものとほぼ同じです。

```
while （条件式） A
```

条件式は Boolean 型である必要があります。while 式は、条件式が true の間、A を評価し続けます。なお、while 式も式なので値を返しますが、while 式には適切な返すべき値がないので Unit 型

の値 () を返します。

Unit 型は Java では void に相当するもので、返すべき値がないときに使われ、唯一の値 () を持ちます。

さて、while 式を使って 1 から 10 までの値を出力してみましょう。

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

scala> while(i <= 10) {
     |   println("i = " + i)
     |   i = i + 1
     | }
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
i = 10
```

Java で while 文を使った場合と同様です。do while 文もありますが、ほぼ Java と同じなので説明は省略します。

練習問題

do while を利用して、0 から数え上げて 9 まで出力して 10 になったらループを終了するメソッド print0To9 を書いてみましょう。loopFrom0To9 は次のような形になります。???の部分を埋めてください。

```
def loop0To9(): Unit = {
  do {
    ???
  } while(???)
}
```

```
def loop0To9(): Unit = {
  var i = 0
  do {
    println(i)
    i += 1
  } while(i < 10)
}
```

```
scala> loop0To9()
0
1
2
3
4
5
6
7
8
9
```

for 式（★★★）

Scala には for 式という制御構文がありますが、これは Java の制御構文と似た使い方ができるものの、全く異なる構文です。for 式の本当の力を理解するには、flatMap, map, withFilter, foreach といったメソッドについて知る必要がありますが、ここでは基本的な for 式の使い方のみを説明します。

for 式の基本的な構文は次のようになります。

```
for(ジェネレータ 1;  ジェネレータ 2; ... ジェネレータ n) A
# ジェネレータ 1 = a1 <- exp1; ジェネレータ 2 = a2 <- exp2; ... ジェネレータ n = an <- expn
```

変数 a1 ‖ an までは好きな名前のループ変数を使うことができます。exp1 から expn までにかける式は一般形を説明するとやっかいなので、さしあたって、ある数の範囲を表す式を使えると覚えておいてください。たとえば、1 to 10 は 1 から 10 まで（10 を含む）の範囲で、1 until 10 は 1 から 10 まで（10 を含まない）の範囲です。

それでは、早速 for 式を使ってみましょう。

```
scala> for(x <- 1 to 5; y <- 1 until 5){
     |   println("x = " + x + " y = " + y)
     | }
x = 1 y = 1
x = 1 y = 2
x = 1 y = 3
x = 1 y = 4
x = 2 y = 1
x = 2 y = 2
x = 2 y = 3
x = 2 y = 4
x = 3 y = 1
x = 3 y = 2
x = 3 y = 3
x = 3 y = 4
```

```
x = 4 y = 1
x = 4 y = 2
x = 4 y = 3
x = 4 y = 4
x = 5 y = 1
x = 5 y = 2
x = 5 y = 3
x = 5 y = 4
```

x を 1 から 5 までループして、y を 1 から 4 までループして x, y の値を出力しています。ここでは、ジェネレータを 2 つだけにしましたが、数を増やせば何重にもループを行うことができます。

for 式の力はこれだけではありません。ループ変数の中から条件にあったものだけを絞り込むこともできます。until の後で if x != y と書いていますが、これは、x と y が異なる値の場合のみを抽出したものです。

```
scala> for(x <- 1 to 5; y <- 1 until 5 if x != y){
     |   println("x = " + x + " y = " + y)
     | }
x = 1 y = 2
x = 1 y = 3
x = 1 y = 4
x = 2 y = 1
x = 2 y = 3
x = 2 y = 4
x = 3 y = 1
x = 3 y = 2
x = 3 y = 4
x = 4 y = 1
x = 4 y = 2
x = 4 y = 3
x = 5 y = 1
x = 5 y = 2
x = 5 y = 3
x = 5 y = 4
```

for 式はコレクションの要素を 1 つ 1 つたどって何かの処理を行うことにも利用することができます。"A", "B", "C", "D", "E"の 5 つの要素からなるリストをたどって全てを出力する処理を書いてみましょう。

```
scala> for(e <- List("A", "B", "C", "D", "E")) println(e)
A
B
C
D
E
```

さらに、for 式はたどった要素を加工して新しいコレクションを作ることもできます。先ほどのリストの要素全てに Pre という文字列を付加してみましょう。

```
scala> for(e <- List("A", "B", "C", "D", "E")) yield {
     |   "Pre" + e
     | }
res9: List[String] = List(PreA, PreB, PreC, PreD, PreE)
```

ここでポイントとなるのは、yield というキーワードです。実は、for 構文は yield キーワードを使うことで、コレクションの要素を加工して返すという全く異なる用途に使うことができます。特に yield キーワードを使った for 式を特別に for-comprehension と呼ぶことがあります。

練習問題

1 から 1000 までの 3 つの整数 a, b, c について、三辺からなる三角形が直角三角形になるような a, b, c の組み合わせを全て出力してください。直角三角形の条件にはピタゴラスの定理を利用してください。ピタゴラスの定理とは三平方の定理とも呼ばれ、a ^ 2 == b ^ 2 + c ^ 2 を満たす、a, b, c の長さの三辺を持つ三角形は、直角三角形になるというものです。

```
for(a <- 1 to 1000; b <- 1 to 1000; c <- 1 to 1000) {
  if(a * a == b * b + c * c) println((a, b, c))
}
```

match 式（★★★）

match 式は使い方によって非常に幅のある制御構造です。match 式の基本構文は

```
マッチ対象の式 match {
  case パターン 1 [if ガード 1] => 式 1
  case パターン 2 [if ガード 2] => 式 2
  case パターン 3 [if ガード 3] => 式 3
  case ...
  case パターン N => 式 N
}
```

のようになりますが、この「パターン」に書ける内容が非常に多岐に渡るためです。まず、Java の switch-case のような使い方をしてみます。たとえば、

```
scala> val taro = "Taro"
taro: String = Taro

scala> taro match {
     |   case "Taro" => "Male"
     |   case "Jiro" => "Male"
     |   case "Hanako" => "Female"
     | }
```

```
res11: String = Male
```

のようにして使うことができます。ここで、taro には文字列"Taro"が入っており、これは case "Taro"にマッチするため、"Male"が返されます。なお、ここで気づいた人もいるかと思いますが、match 式も値を返します。match 式の値は、マッチしたパターンの=>の右辺の式を評価したものになります。

パターンは文字列だけでなく数値など任意の値を扱うことができます：

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

scala> one match {
     |   case 1 => "one"
     |   case 2 => "two"
     |   case _ => "other"
     | }
res12: String = one
```

ここで、パターンの箇所に_が出てきましたが、これは switch-case の default のようなもので、あらゆるものにマッチするパターンです。このパターンをワイルドカードパターンと呼びます。match 式を使うときは、漏れがないようにするために、ワイルドカードパターンを使うことが多いです。

パターンをまとめる

Java や C などの言語で switch-case 文を学んだ方には、Scala のパターンマッチがいわゆるフォールスルー（fall through）の動作をしないことに違和感があるかもしれません。

```
"abc" match {
  case "abc" => println("first")   // ここで処理が終了
  case "def" => println("second") // こっちは表示されない
}
```

C 言語の switch-case 文のフォールスルー動作は利点よりバグを生み出すことが多いということで有名なものでした。Java が C 言語のフォールスルー動作を引き継いだことはしばしば非難されます。それで Scala のパターンマッチにはフォールスルー動作がないわけですが、複数のパターンをまとめたいときのために|があります

```
"abc" match {
  case "abc" | "def" =>
    println("first")
    println("second")
}
```

パターンマッチによる値の取り出し

switch-case 以外の使い方としては、コレクションの要素の一部にマッチさせる使い方があります。次のプログラムを見てみましょう。

```
scala> val lst = List("A", "B", "C", "D", "E")
lst: List[String] = List(A, B, C, D, E)

scala> lst match {
     |   case List("A", b, c, d, e) =>
     |     println("b = " + b)
     |     println("c = " + c)
     |     println("d = " + d)
     |     println("e = " + e)
     |   case _ =>
     |     println("nothing")
     | }
b = B
c = C
d = D
e = E
```

ここでは、List の先頭要素が"A"で 5 要素のパターンにマッチすると、残りの b, c, d, e に List の 2 番目以降の要素が束縛されて、=>の右辺の式が評価されることになります。match 式では、特にコレクションの要素にマッチさせる使い方が頻出します。

パターンマッチではガード式を用いて、パターンにマッチして、かつ、ガード式（Boolean 型でなければならない）にもマッチしなければ右辺の式が評価されないような使い方もできます。

```
scala> val lst = List("A", "B", "C", "D", "E")
lst: List[String] = List(A, B, C, D, E)

scala> lst match {
     |   case List("A", b, c, d, e) if b != "B" =>
     |     println("b = " + b)
     |     println("c = " + c)
     |     println("d = " + d)
     |     println("e = " + e)
     |   case _ =>
     |     println("nothing")
     | }
nothing
```

ここでは、パターンマッチのガード条件に、List の 2 番目の要素が"B"でないこと、という条件を指定したため、最初の条件にマッチせず_にマッチしたのです。

また、パターンマッチのパターンはネストが可能です。先ほどのプログラムを少し改変して、先頭が List("A") であるような List にマッチさせてみましょう。

```
scala> val lst = List(List("A"), List("B", "C", "D", "E"))
lst: List[List[String]] = List(List(A), List(B, C, D, E))

scala> lst match {
     |   case List(a@List("A"), x) =>
     |   println(a)
     |   println(x)
     |   case _ => println("nothing")
     | }
List(A)
List(B, C, D, E)
```

lst は List("A") と List("B", "C", "D", "E") の 2 要素からなる List です。ここで、match 式を使うことで、先頭が List("A") であるというネストしたパターンを記述できていることがわかります。また、パターンの前に@がついているのは as パターンと呼ばれるもので、@の後に続くパターンにマッチする式を@の前の変数（ここでは a）に束縛します。as パターンはパターンが複雑なときにパターンの一部だけを切り取りたい時に便利です。

型によるパターンマッチ

パターンとしては値が特定の型に所属する場合にのみマッチするパターンも使うことができます。値が特定の型に所属する場合にのみマッチするパターンは、**名前:マッチする型**の形で使います。たとえば、以下のようにして使うことができます。なお、AnyRef 型は、Java の Object 型に相当する型で、あらゆる参照型の値を AnyRef 型の変数に格納することができます。

```
scala> import java.util.Locale
import java.util.Locale

scala> val obj: AnyRef = "String Literal"
obj: AnyRef = String Literal

scala> obj match {
     |   case v:java.lang.Integer =>
     |     println("Integer!")
     |   case v:String =>
     |     println(v.toUpperCase(Locale.ENGLISH))
     | }
STRING LITERAL
```

java.lang.Integer にはマッチせず、String にマッチしていることがわかります。このパターンは例外処理や equals の定義などで使うことがあります。型でマッチした値は、その型にキャストしたのと同じように扱うことができます。たとえば、上記の式で String 型にマッチした v は String 型のメソッドである toUpperCase を呼びだすことができます。しばしば Scala ではキャストの代わりにパターンマッチが用いられるので覚えておくとよいでしょう。

JVM の制約による型のパターンマッチの落とし穴

ただし、型のパターンマッチで注意しなければならないことが 1 つあります。Scala を実行する JVM の制約により、型変数を使った場合、正しくパターンマッチがおこなわれません。

たとえば、以下の様なパターンマッチを REPL で実行しようとすると、警告が出てしまいます。

```
scala> val obj: Any = List("a")
obj: Any = List(a)

scala> obj match {
     |   case v: List[Int]    => println("List[Int]")
     |   case v: List[String] => println("List[String]")
     | }
<console>:16: warning: non-variable type argument Int in type pattern List[Int] (the underlying of List[Int]) is unche
         case v: List[Int]    => println("List[Int]")
                 ^
<console>:17: warning: non-variable type argument String in type pattern List[String] (the underlying of List[String]
         case v: List[String] => println("List[String]")
                 ^
<console>:17: warning: unreachable code
         case v: List[String] => println("List[String]")
                                         ^
List[Int]
```

型としては `List[Int]` と `List[String]` は違う型なのですが、パターンマッチではこれを区別できません。

最初の 2 つの警告の意味は Scala コンパイラの「型消去」という動作により `List[Int]` の `Int` の部分が消されてしまうのでチェックされないということです。

結果的に 2 つのパターンは区別できないものになり、パターンマッチは上から順番に実行されていくので、2 番目のパターンは到達しないコードになります。3 番目の警告はこれを意味しています。

型変数を含む型のパターンマッチは、

```
obj match {
  case v: List[_] => println("List[_]")
}
```

このようにワイルドカードパターンを使うとよいでしょう。

練習問題

```
new scala.util.Random(new java.security.SecureRandom()).alphanumeric.take(5).toList
```

以上のコードを利用して、最初と最後の文字が同じアルファベットであるランダムな 5 文字の文字列を 1000 回出力してください。`new scala.util.Random(new java.security.SecureRandom()).alphanumeric.take(` という値は、呼びだす度にランダムな 5 個の文字（`Char` 型）のリストを与えます。なお、以上のコー

ドで生成されたリストの一部分を利用するだけでよく、最初と最後の文字が同じアルファベットであるリストになるまで試行を続ける必要はありません。これは、List(a, b, d, e, f) が得られた場合に、List(a, b, d, e, a) のようにしても良いということです。

```
for(i <- 1 to 1000) {
  val s = new scala.util.Random(new java.security.SecureRandom()).alphanumeric.take(5).toList match {
    case List(a,b,c,d,_) => List(a,b,c,d,a).mkString
  }
  println(s)
}
```

ここまで書いただけでも、match 式は switch-case に比べてかなり強力であることがわかると思います。ですが、match 式の力はそれにとどまりません。後述しますが、パターンには自分で作ったクラス（のオブジェクト）を指定することでさらに強力になります。

第 7 章

クラス

これからクラスとは何かということに関して説明してます。最低限、Java のクラスがわかっている事を前提にしますが、ご了承ください。

クラス定義（★★★）

Scala におけるクラスは、記法を除けば Java 言語のクラスと大して変わりません。Scala のクラス定義はおおまかには次のような形を取ります。

```
class クラス名（コンストラクタ引数 1 :コンストラクタ引数 1 の型，コンストラクタ引数 2 :コンストラクタ引数 2 の型，...）{
  0 個以上のフィールドの定義またはメソッド定義
}
```

たとえば、点を表すクラス Point を定義したいとします。Point は x 座標を表すフィールド x（Int 型）とフィールド y（Int 型）からなるとします。このクラス Point を Scala で書くと次のようになります。

```
class Point(_x: Int, _y: Int) {
  val x = _x
  val y = _y
}
```

コンストラクタの引数をそのまま公開したい場合は、以下のように短く書くこともできます。

```
class Point(val x: Int, val y: Int)
```

- クラス名の直後にコンストラクタの定義がある
- val/var によって、コンストラクタ引数をフィールドとして公開することができる

点に注目してください。まず、最初の点ですが、Scala では 1 クラスに付き、基本的には 1 つのコンストラクタしか使いません。文法上は複数のコンストラクタを定義できるようになっていますが、実際に使うことはまずないので覚える必要はないでしょう。一応、Scala では複数のコンストラクタが定義できるので、この最初の 1 つのコンストラクタをプライマリコンストラクタとして特別に扱って

います。

プライマリコンストラクタの引数に val/var をつけるとそのフィールドは公開され、外部からアクセスできるようになります。なお、コンストラクタ引数のスコープはクラス定義全体におよびます。そのため、

```
class Point(val x: Int, val y: Int) {
  def +(p: Point): Point = {
    new Point(x + p.x, y + p.y)
  }
  override def toString(): String = "(" + x + ", " + y + ")"
}
```

のように、メソッド定義の中から直接コンストラクタ引数を参照できます。

メソッド定義（★★★）

先ほど既にメソッド定義の例として + メソッドの定義が出てきましたが、一般的には、

```
(private[this]/protected[package 名]) def メソッド名 (引数名 1: 引数 1 の型, 引数名 2: 引数 2 の型, ...): 返り値の型 = {
  本体のコード
}
```

という形をとります。ちなみに、メソッド本体が 1 つの式だけからなる場合は、

```
(private[this]/protected[package 名]) def メソッド名 (引数名 1: 引数 1 の型, 引数名 2: 引数 2 の型, ...): 返り値の型 = 本体のコー
```

と書けます（実際には、こちらの方が基本形で、= {}を使ったスタイルの方が{}内に複数の式を並べて書けることを利用した派生形になりますが、前者のパターンを使うことが多いでしょう）。

返り値の型は省略しても特別な場合以外型推論してくれますが、読みやすさのために、返り値の型は明記する習慣を付けるようにしましょう。また、private を付けるとそのクラス内だけから、protected を付けるとそのクラスの派生クラスからのみアクセスできるフィールドになります。さらに、private[this] をつけると、同じオブジェクトからのみ、protected[パッケージ名] をつけると、追加で同じパッケージに所属しているもの全てからアクセスできるようになります。private も protected も付けない場合、そのフィールドは public とみなされます。

先ほど定義した Point クラスを REPL から使ってみましょう。

```
scala> class Point(val x: Int, val y: Int) {
     |   def +(p: Point): Point = {
     |     new Point(x + p.x, y + p.y)
     |   }
     |   override def toString(): String = "(" + x + ", " + y + ")"
     | }
defined class Point

scala> val p1 = new Point(1, 1)
```

```
p1: Point = (1, 1)

scala> val p2 = new Point(2, 2)
p2: Point = (2, 2)

scala> p1 + p2
res0: Point = (3, 3)
```

メソッドのカリー化（★★★）

メソッドは複数の引数リストを持つことができます。メソッドが

```
def メソッド名(引数名1: 型名1, 引数名2: 型名2, ...)(引数名N: 型名N, ..., 引数M: N): 返り値型 = 本体
```

のように複数の引数リストを持つように定義したとき、メソッド定義はカリー化されていると言います。カリー化は普通の関数型言語では重要なテクニックですが、Scala でのカリー化は意味合いがいささかことなります。後述する implicit parameter などを使う場合、複数の引数リストを作らなければならないためカリー化が必要になるのです。何はともあれ、カリー化された加算メソッドを定義してみましょう。

```
scala> class Adder {
     |   def add(x: Int)(y: Int): Int = x + y
     | }
defined class Adder

scala> val adder = new Adder()
adder: Adder = Adder@b6b5c42

scala> adder.add(2)(3)
res1: Int = 5

scala> adder.add(2) _
res2: Int => Int = <function1>
```

カリー化されたメソッドは `obj.m(x, y)` の形式でなく `obj.m(x)(y)` の形式で呼びだすことになります。また、一番下の例のように最初の引数だけを適用して新しい関数を作る（部分適用）こともできます。

フィールド定義（★★★）

フィールド定義は

```
(private/protected) (val/var) フィールド名: フィールドの型 = フィールドに代入される値の式
```

という形を取ります。`val` の場合は変更不能、`var` の場合は変更可能なフィールドになります。また、`private` を付けるとそのクラス内だけから、`protected` を付けるとそのクラスの派生クラスか

らのみアクセスできるフィールドになります。`private` も `protected` も付けない場合、そのフィールドは public とみなされます。

継承（★★★）

クラスのもう 1 つの機能は、継承です。 継承には 2 つの目的があります。1 つは継承によりスーパークラスの実装をサブクラスでも使うことで実装を再利用することです。もう 1 つは複数のサブクラスが共通のスーパークラスのインタフェースを継承することで処理を共通化することです[*1]。

実装の継承には複数の継承によりメソッドやフィールドの名前が衝突する場合の振舞いなどに問題があることが知られており、Java では実装継承が 1 つだけに限定されています。Java 8 ではインタフェースにデフォルトの実装を持たせられるようになりましたが、変数は持たせられないなどの制約があります。Scala ではトレイトという仕組みで複数の実装の継承を実現していますが、トレイトについては別の節で説明します。

ここでは通常の Scala のクラスの継承について説明します。Scala でのクラスの継承は次のような構文になります。

```
class (....) extends 継承元クラス {
  ....
}
```

基本的に、継承のはたらきは Java のクラスと同じですが、既存のメソッドをオーバーライドするときは `override` キーワードを使わなければならない点が異なります。たとえば、

```
scala> class APrinter() {
     |   def print(): Unit = {
     |     println("A")
     |   }
     | }
defined class APrinter

scala> class BPrinter() extends APrinter {
     |   override def print(): Unit = {
     |     println("B")
     |   }
     | }
defined class BPrinter

scala> new APrinter().print
A

scala> new BPrinter().print
B
```

[*1] このように継承などにより型に親子関係を作り、複数の型に共通のインタフェースを持たせることをサブタイピング・ポリモーフィズムと呼びます。Scala では他にも構造的部分型というサブタイピング・ポリモーフィズムの機能がありますが、実際に使われることが少ないため、このテキストでは説明を省略しています。

のようにすることができます。ここで override キーワードをはずすと、

```
scala> class BPrinter() extends APrinter {
     |   def print(): Unit = {
     |     println("B")
     |   }
     | }
<console>:14: error: overriding method print in class APrinter of type ()Unit;
 method print needs `override' modifier
         def print(): Unit = {
             ^
```

のようにメッセージを出力して、**コンパイルエラー**になります。Java ではしばしば、気付かずに既存のメソッドをオーバーライドするつもりで新しいメソッドを定義してしまうというミスがありましたが、Scala では override キーワードを使って言語レベルでこの問題に対処しているのです。

第 8 章

object（★★★）

Scala では、全ての値がオブジェクトです。また、全てのメソッドは何らかのオブジェクトに所属しています。そのため、Java のようにクラスに属する static フィールドや static メソッドといったものを作成することができません。その代わりというと若干語弊があるのですが、`object` キーワードによって、同じ名前のシングルトンオブジェクトをグローバルな名前空間に 1 つ定義することができます。`object` キーワードによって定義したオブジェクトもオブジェクトであるため、メソッドやフィールドを定義することができます。

`object` 構文の主な用途としては、

- ユーティリティメソッドやグローバルな状態の置き場所（Java で言う static メソッドやフィールド）
- オブジェクトのファクトリメソッド
- Singleton パターン

3 つが挙げられます。とはいえ、Singleton パターンを実現するために使われることはほとんどなく、もっぱら最初の 2 つの用途で使われます。

object の基本構文はクラスとほとんど同じで、

```
object オブジェクト名 extends クラス名 with トレイト名 1 with トレイト名 2 ... {
  本体
}
```

のようになります。Scala では標準で `Predef` という `object` が定義・インポートされており、これは最初の使い方に当てはまります。たとえば、println() などとなにげなく使っていたメソッドも実は Predef object のメソッドなのです。

一方、2 番めの使い方について考えてみます。点を表す Point クラスのファクトリを `object` で作ろうとすると、次のようになります。apply という名前のメソッドは Scala 処理系によって特別に扱われ、`Point(x)` のような記述があった場合で、Point `object` に `apply` という名前のメソッドが定義されていた場合、`Point.apply(x)` と解釈されます。これを利用して Point object の `apply` メソッ

ドでオブジェクトを生成するようにすることで、Point(3, 5) のような記述でオブジェクトを生成できるようになります。

```
scala> class Point(val x:Int, val y:Int)
defined class Point

scala> object Point {
     |   def apply(x: Int, y: Int): Point = new Point(x, y)
     | }
defined object Point
warning: previously defined class Point is not a companion to object Point.
Companions must be defined together; you may wish to use :paste mode for this.
```

これは、new Point() で直接 Point オブジェクトを生成するのに比べて、

- クラス（Point）の実装詳細を内部に隠しておける（インタフェースのみを外部に公開する）
- Point ではなく、そのサブクラスのインスタンスを返すことができる

といったメリットがあります。なお、上記の記述はケースクラスを用いてもっと簡単に

```
scala> case class Point(x: Int, y: Int)
defined class Point
```

と書けます。ケースクラスは後述するパターンマッチのところでも出てきますが、ここではその使い方については触れません。簡単に言うとケースクラスは、それをつけたクラスのプライマリコンストラクタ全てのフィールドを公開し、equqls()・hashCode()・toString() などのオブジェクトの基本的なメソッドを持ったクラスを生成し、また、そのクラスのインスタンスを生成するためのファクトリメソッドを生成するものです。たとえば、case class Point(x: Int, y: Int) で定義した Point クラスは equals() メソッドを明示的に定義してはいませんが、

```
Point(1, 2).equals(Point(1, 2))
```

を評価した値は true になります。

コンパニオンオブジェクト（★★★）

クラス名と同じ名前の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呼ばれます。コンパニオンオブジェクトは対応するクラスに対して特権的なアクセス権を持っています。たとえば、weight を private にした場合、

```
class Person(name: String, age: Int, private val weight: Int)

object Hoge {
  val taro = new Person("Taro", 20, 70)
  println(taro.weight)
}
```

は NG ですが、

```
class Person(name: String, age: Int, private val weight: Int)

object Person {
  val taro = new Person("Taro", 20, 70)
  println(taro.weight)
}
```

は OK です。なお、コンパニオンオブジェクトでも、`private[this]`（そのオブジェクト内からのみアクセス可能）なクラスのメンバーに対してはアクセスできません。単に `private` とした場合、コンパニオンオブジェクトからアクセスできるようになります。

上記のような、コンパニオンオブジェクトを使ったコードを REPL で試す場合は、REPL の `:paste` コマンドを使って、クラスとコンパニオンオブジェクトを一緒にペーストするようにしてください。クラスとコンパニオンオブジェクトは同一ファイル中に置かれていなければならないのですが、REPL で両者を別々に入力した場合、コンパニオン関係を REPL が正しく認識できないのです。

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

class Person(name: String, age: Int, private val weight: Int)

object Person {
  val taro = new Person("Taro", 20, 70)
  println(taro.weight)
}

// Exiting paste mode, now interpreting.

defined class Person
defined object Person
```

練習問題

クラスを定義して、そのクラスのコンパニオンオブジェクトを定義してみましょう。コンパニオンオブジェクトが同名のクラスに対する特権的なアクセス権を持っていることを、クラスのフィールドを `private` にして、そのフィールドへアクセスできることを通じて確認してみましょう。また、クラスのフィールドを `private[this]` にして、そのフィールドへアクセスできないことを確認してみましょう。

第 9 章

トレイト

私たちの作るプログラムはしばしば数万行、多くなると数十万行やそれ以上に及ぶことがあります。その全てを一度に把握することは難しいので、プログラムを意味のあるわかりやすい単位で分割しなければなりません。さらに、その分割された部品はなるべく柔軟に組み立てられ、大きなプログラムを作れると良いでしょう。

プログラムの分割（モジュール化）と組み立て（合成）は、オブジェクト指向プログラミングでも関数型プログラミングにおいても重要な設計の概念になります。そして、Scala のオブジェクト指向プログラミングにおけるモジュール化の中心的な概念になるのがトレイトです。

この節では Scala のトレイトの機能を一通り見ていきましょう。

トレイトの基本（★★★）

Scala のトレイトはクラスに比べて以下のような特徴があります。

- 複数のトレイトを 1 つのクラスやトレイトにミックスインできる
- 直接インスタンス化できない
- クラスパラメータ（コンストラクタの引数）を取ることができない

以下、それぞれの特徴の紹介をしていきます。

複数のトレイトを 1 つのクラスやトレイトにミックスインできる

Scala のトレイトはクラスとは違い、複数のトレイトを 1 つのクラスやトレイトにミックスインすることができます。

```
trait TraitA

trait TraitB

class ClassA
```

```
class ClassB

// コンパイルできる
class ClassC extends ClassA with TraitA with TraitB

// コンパイルエラー！
// class ClassB needs to be a trait to be mixed in
class ClassD extends ClassA with ClassB
```

上記の例では ClassA と TraitA と TraitB を継承した ClassC を作ることはできますが ClassA と ClassB を継承した ClassD は作ることができません。「class ClassB needs to be a trait to be mixed in」というエラーメッセージが出ますが、これは「ClassB をミックスインさせるためにはトレイトにする必要がある」という意味です。複数のクラスを継承させたい場合はクラスをトレイトにしましょう。

直接インスタンス化できない

Scala のトレイトはクラスと違い、直接インスタンス化できません。

```
trait TraitA

object ObjectA {
  // コンパイルエラー！
  // trait TraitA is abstract; cannot be instantiated
  val a = new TraitA
}
```

この制限は回避する方法がいくつかあります。1 つはインスタンス化できるようにトレイトを継承したクラスを作ることです。もう 1 つはトレイトに実装を与えてインスタンス化する方法です。

```
trait TraitA

class ClassA extends TraitA

object ObjectA {
  // クラスにすればインスタンス化できる
  val a = new ClassA

  // 実装を与えてもインスタンス化できる
  val a2 = new TraitA {}
}
```

このように実際使う上では、あまり問題にならない制限でしょう。

クラスパラメータ（コンストラクタの引数）を取ることができない

Scala のトレイトはクラスと違いパラメータ（コンストラクタの引数）を取ることができないという制限があります[*1]。

```
// 正しいプログラム
class ClassA(name: String) {
  def printName() = println(name)
}

// コンパイルエラー！
// traits or objects may not have parameters
trait TraitA(name: String) {
  def printName: Unit = println(name)
}
```

これもあまり問題になることはありません。トレイトに抽象メンバーを持たせることで値を渡すことができます。インスタンス化できない問題のときと同じようにクラスに継承させたり、インスタンス化のときに抽象メンバーを実装をすることでトレイトに値を渡すことができます。

```
trait TraitA {
  val name: String
  def printName(): Unit = println(name)
}

// クラスにして name を上書きする
class ClassA(val name: String) extends TraitA

object ObjectA {
  val a = new ClassA("dwango")

  // name を上書きするような実装を与えてもよい
  val a2 = new TraitA { val name = "kadokawa" }
}
```

以上のようにトレイトの制限は実用上ほとんど問題にならないようなものであり、その他の点ではクラスと同じように使うことができます。つまり実質的に多重継承と同じようなことができるわけです。そしてトレイトのミックスインはモジュラリティに大きな恩恵をもたらします。是非使いこなせるようになりましょう。

「トレイト」という用語について（★）

この節では、トレイトやミックスインなどオブジェクトの指向の用語が用いられますが、他の言語などで用いられる用語とは少し違う意味を持つかもしれないので、注意が必要です。

[*1] 将来のバージョンでは、パラメータを取れるようになるかもしれないという話があります http://docs.scala-lang.org/sips/pending/trait-parameters.html

トレイトは Schärli らによる 2003 年の ECOOP に採択された論文『Traits: Composable Units of Behaviour』がオリジナルとされていますが、この論文中のトレイトの定義と Scala のトレイトの仕様は、合成時の動作や、状態変数の取り扱いなどについて、異なっているように見えます。

しかし、トレイトやミックスインという用語は言語によって異なるものであり、我々が参照している Scala の公式ドキュメントや『Scala スケーラブルプログラミング』でも「トレイトをミックスインする」という表現が使われていますので、ここではそれに倣いたいと思います。

トレイトの様々な機能

菱形継承問題（★★）

以上見てきたようにトレイトはクラスに近い機能を持ちながら実質的な多重継承が可能であるという便利なものなのですが、1 つ考えなければならないことがあります。多重継承を持つプログラミング言語が直面する「菱形継承問題」というものです。

以下のような継承関係を考えてみましょう。`greet` メソッドを定義した `TraitA` と、`greet` を実装した `TraitB` と `TraitC`、そして `TraitB` と `TraitC` のどちらも継承した `ClassA` です。

```
trait TraitA {
  def greet(): Unit
}

trait TraitB extends TraitA {
  def greet(): Unit = println("Good morning!")
}

trait TraitC extends TraitA {
  def greet(): Unit = println("Good evening!")
}

class ClassA extends TraitB with TraitC
```

`TraitB` と `TraitC` の `greet` メソッドの実装が衝突しています。この場合 `ClassA` の `greet` はどのような動作をすべきなのでしょうか？ `TraitB` の `greet` メソッドを実行すべきなのか、`TraitC` の `greet` メソッドを実行すべきなのか。多重継承をサポートする言語はどれもこのようなあいまいさの問題を抱えており、対処が求められます。

ちなみに、上記の例を Scala でコンパイルすると以下のようなエラーが出ます。

```
ClassA.scala:13: error: class ClassA inherits conflicting members:
  method greet in trait TraitB of type ()Unit  and
  method greet in trait TraitC of type ()Unit
(Note: this can be resolved by declaring an override in class ClassA.)
class ClassA extends TraitB with TraitC
      ^
```

```
one error found
```

Scala では override 指定なしの場合メソッド定義の衝突はエラーになります。

この場合の 1 つの解法は、コンパイルエラーに「Note: this can be resolved by declaring an override in class ClassA.」とあるように ClassA で greet を override することです。

```
class ClassA extends TraitB with TraitC {
  override def greet(): Unit = println("How are you?")
}
```

このとき ClassA で super に型を指定してメソッドを呼びだすことで、TraitB や TraitC のメソッドを指定して使うこともできます。

```
class ClassB extends TraitB with TraitC {
  override def greet(): Unit = super[TraitB].greet()
}
```

実行結果は以下にようになります。

```
scala> (new ClassA).greet()
How are you?

scala> (new ClassB).greet()
Good morning!
```

では、TraitB と TraitC の両方のメソッドを呼び出したい場合はどうでしょうか？ 1 つの方法は上記と同じように TraitB と TraitC の両方のクラスを明示して呼びだすことです。

```
class ClassA extends TraitB with TraitC {
  override def greet(): Unit = {
    super[TraitB].greet()
    super[TraitC].greet()
  }
}
```

しかし、継承関係が複雑になった場合にすべてを明示的に呼ぶのは大変です。また、コンストラクタのように必ず呼び出されるメソッドもあります。

Scala のトレイトにはこの問題を解決するために「線形化（linearization）」という機能があります。

線形化（linearization）（★）

Scala のトレイトの線形化機能とは、トレイトがミックスインされた順番をトレイトの継承順番と見做すことです。

次に以下の例を考えてみます。先程の例との違いは TraitB と TraitC の greet メソッド定義に override 修飾子が付いていることです。

```
trait TraitA {
  def greet(): Unit
}

trait TraitB extends TraitA {
  override def greet(): Unit = println("Good morning!")
}

trait TraitC extends TraitA {
  override def greet(): Unit = println("Good evening!")
}

class ClassA extends TraitB with TraitC
```

この場合はコンパイルエラーにはなりません。では ClassA の greet メソッドを呼び出した場合、いったい何が表示されるのでしょうか？ 実際に実行してみましょう。

```
scala> (new ClassA).greet()
Good evening!
```

ClassA の greet メソッドの呼び出しで、TraitC の greet メソッドが実行されました。これはトレイトの継承順番が線形化されて、後からミックスインした TraitC が優先されているためです。つまりトレイトのミックスインの順番を逆にすると TraitB が優先されるようになります。以下のようにミックスインの順番を変えてみます。

```
class ClassB extends TraitC with TraitB
```

すると ClassB の greet メソッドの呼び出して、今度は TraitB の greet メソッドが実行されます。

```
scala> (new ClassB).greet()
Good morning!
```

super を使うことで線形化された親トレイトを使うこともできます

```
trait TraitA {
  def greet(): Unit = println("Hello!")
}

trait TraitB extends TraitA {
  override def greet(): Unit = {
    super.greet()
    println("My name is Terebi-chan.")
  }
}

trait TraitC extends TraitA {
  override def greet(): Unit = {
```

```
    super.greet()
    println("I like niconico.")
  }
}

class ClassA extends TraitB with TraitC
class ClassB extends TraitC with TraitB
```

この greet メソッドの結果もまた継承された順番で変わります。

```
scala> (new ClassA).greet()
Hello!
My name is Terebi-chan.
I like niconico.

scala> (new ClassB).greet()
Hello!
I like niconico.
My name is Terebi-chan.
```

線形化の機能によりミックスインされたすべてのトレイトの処理を簡単に呼び出せるようになりました。このような線形化によるトレイトの積み重ねの処理を Scala の用語では積み重ね可能なトレイト（Stackable Trait）と呼ぶことがあります。

この線形化が Scala の菱形継承問題に対する対処法になるわけです。

abstract override（★）

通常のメソッドのオーバーライドで super を使ってスーパークラスのメソッドを呼びだす場合、当然のことながら継承元のスーパークラスにそのメソッドの実装がなければならないわけですが、Scala には継承元のスーパークラスにそのメソッドの実装がない場合でもメソッドのオーバーライドが可能な abstract override という機能があります。

abstract override ではない override と abstract override を比較してみましょう。

```
trait TraitA {
  def greet(): Unit
}

// コンパイルエラー！
// method greet in trait TraitA is accessed from super. It may not be abstract unless it is overridden by a member dec
trait TraitB extends TraitA {
  override def greet(): Unit = {
    super.greet()
    println("Good morning!")
  }
}

// コンパイルできる
trait TraitC extends TraitA {
```

```
  abstract override def greet(): Unit = {
    super.greet()
    println("Good evening!")
  }
}
```

abstract 修飾子を付けていない TraitB はコンパイルエラーになってしまいました。エラーメッセージの意味は、TraitA の greet メソッドには実装がないので abstract override を付けない場合オーバーライドが許されないということです。

オーバーライドを abstract override にすることでスーパークラスのメソッドの実装がない場合でもオーバーライドすることができます。この特性は抽象クラスに対しても積み重ねの処理が書けるということを意味します。

しかし abstract override でも 1 つ制約があり、ミックスインされてクラスが作られるときにはスーパークラスのメソッドが実装されてなければなりません。

```
trait TraitA {
  def greet(): Unit
}

trait TraitB extends TraitA {
  def greet(): Unit =
    println("Hello!")
}

trait TraitC extends TraitA {
  abstract override def greet(): Unit = {
    super.greet()
    println("I like niconico.")
  }
}

// コンパイルエラー！
// class ClassA needs to be a mixin, since method greet in trait TraitC of type ()Unit is marked `abstract' and `overr
class ClassA extends TraitC

// コンパイルできる
class ClassB extends TraitB with TraitC
```

自分型（★★）

Scala にはクラスやトレイトの中で自分自身の型にアノテーションを記述することができる機能があります。これを自分型アノテーション（self type annotations）や単に自分型（self types）などと呼びます。

例を見てみましょう。

```
trait Greeter {
  def greet(): Unit
}

trait Robot {
  self: Greeter =>

  def start(): Unit = greet()
}
```

このロボット（Robot）は起動（start）するときに挨拶（greet）するようです。Robot は直接 Greeter を継承していないのにもかかわらず greet メソッドを使えていることに注意してください。

このロボットのオブジェクトを実際に作るためには greet メソッドを実装したトレイトが必要になります。REPL を使って動作を確認してみましょう。

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

trait HelloGreeter extends Greeter {
  def greet(): Unit = println("Hello!")
}

// Exiting paste mode, now interpreting.

defined trait HelloGreeter

scala> val r = new Robot with HelloGreeter
r: Robot with HelloGreeter = $anon$1@1e5756c0

scala> r.start()
Hello!
```

自分型を使う場合は、抽象トレイトを指定し、後から実装を追加するという形になります。このように後から（もしくは外から）利用するモジュールの実装を与えることを依存性の注入（Dependency Injection）と呼ぶことがあります。自分型を使われている場合、この依存性の注入のパターンが使われていると考えてよいでしょう。

ではこの自分型によるトレイトの指定は以下のように直接継承する場合と比べてどのような違いがあるのでしょうか。

```
trait Greeter {
  def greet(): Unit
}

trait Robot2 extends Greeter {
  def start(): Unit = greet()
}
```

オブジェクトを生成するという点では変わりません。Robot2 も先程と同じように作成することができます。ただし、このトレイトを利用する側や、継承したトレイトやクラスには Greeter トレイト

の見え方に違いができます。

```
scala> val r: Robot = new Robot with HelloGreeter
r: Robot = $anon$1@10470bfa

scala> r.greet()
<console>:9: error: value greet is not a member of Robot
              r.greet()

scala> val r: Robot2 = new Robot2 with HelloGreeter
r: Robot = $anon$1@10470bfa

scala> r.greet()
Hello!
```

継承で作られた Robot2 オブジェクトでは Greeter トレイトの greet メソッドを呼び出せてしまいますが、自分型で作られた Robot オブジェクトでは greet メソッドを呼びだすことができません。

Robot が利用を宣言するためにある Greeter のメソッドが外から呼び出せてしまうことはあまり良いことではありません。この点で自分型を使うメリットがあると言えるでしょう。逆に単に依存性を注入できればよいという場合には、この動作は煩わしく感じられるかもしれません。

もう 1 つ自分型の特徴としては型の循環参照を許す点です。

自分型を使う場合は以下のようなトレイトの相互参照を許しますが、

```
// コンパイルできる
trait Greeter {
  self: Robot =>

  def greet(): Unit = println(s"My name is $name")
}

trait Robot {
  self: Greeter =>

  def name: String

  def start(): Unit = greet()
}
```

これを先ほどのように継承に置き換えることではできません。

```
// コンパイルエラー
// illegal cyclic reference involving trait Greeter
trait Greeter extends Robot {
  def greet(): Unit = println(s"My name is $name")
}

trait Robot extends Greeter {
  def name: String
```

```
  def start(): Unit = greet()
}
```

しかし、このように循環するような型構成を有効に使うのは難しいかもしれません。

依存性の注入を使う場合、継承を使うか、自分型を使うかというのは若干悩ましい問題かもしれません。機能的には継承があればよいと言えますが、上記のような可視性の問題がありますし、自分型を使うことで依存性の注入を利用しているとわかりやすくなる効果もあります。利用する場合はチームで相談するとよいかもしれません。

落とし穴：トレイトの初期化順序（★★）

Scala のトレイトの val の初期化順序はトレイトを使う上で大きな落とし穴になります。以下のような例を考えてみましょう。トレイト A で変数 foo を宣言し、トレイト B が foo を使って変数 bar を作成し、クラス C で foo に値を代入してから bar を使っています。

```
trait A {
  val foo: String
}

trait B extends A {
  val bar = foo + "World"
}

class C extends B {
  val foo = "Hello"

  def printBar(): Unit = println(bar)
}
```

REPL でクラス C の printBar メソッドを呼び出してみましょう。

```
scala> (new C).printBar()
nullWorld
```

nullWorld と表示されてしまいました。クラス C で foo に代入した値が反映されていないようです。どうしてこのようなことが起きるかというと、Scala のクラスおよびトレイトはスーパークラスから順番に初期化されるからです。この例で言えば、クラス C はトレイト B を継承し、トレイト B はトレイト A を継承しています。つまり初期化はトレイト A が一番先におこなわれ、変数 foo が宣言され、中身は何も代入されていないので、null になります。次にトレイト B で変数 bar が宣言され null である foo と “World” という文字列から"nullWorld"という文字列が作られ、変数 bar に代入されます。先ほど表示された文字列はこれになります。

このような簡単な例なら気づきやすいのですが、似たような形の大規模な例もあります。先ほど自分型で紹介した「依存性の注入」は、上位のトレイトで宣言したものを、中間のトレイトで使い、最終的にインスタンス化するときにミックスインするという手法です。ここでもうっかりすると同じよ

うな罠を踏んでしまいます。Scala 上級者でもやってしまうのが val の初期化順の罠なのです。

トレイトの val の初期化順序の回避方法（★★）

では、この罠はどうやれば回避できるのでしょうか。上記の例で言えば、使う前にちゃんと foo が初期化されるように、bar の初期化を遅延させることです。処理を遅延させるには lazy val か def を使います。

具体的なコードを見てみましょう。

```
trait A {
  val foo: String
}

trait B extends A {
  lazy val bar = foo + "World" // もしくは def bar でもよい
}

class C extends B {
  val foo = "Hello"

  def printBar(): Unit = println(bar)
}
```

先ほどの nullWorld が表示されてしまった例と違い、bar の初期化に lazy val が使われるようになりました。これにより bar の初期化が実際に使われるまで遅延されることになります。その間にクラス C で foo が初期化されることにより、初期化前の foo が使われることがなくなるわけです。

今度はクラス C の printBar メソッドを呼び出してもちゃんと HelloWorld と表示されます。

```
scala> (new C).printBar()
HelloWorld
```

lazy val は val に比べて若干処理が重く、複雑な呼び出しでデッドロックが発生する場合があります。val のかわりに def を使うと毎回値を計算してしまうという問題があります。しかし、両方とも大きな問題にならない場合が多いので、特に val の値を使って val の値を作り出すような場合は lazy val か def を使うことを検討しましょう。

トレイトの val の初期化順序を回避するもう 1 つの方法としては事前定義（Early Definitions）を使う方法もあります。事前定義というのはフィールドの初期化をスーパークラスより先におこなう方法です。

```
trait A {
  val foo: String
}

trait B extends A {
  val bar = foo + "World" // val のままでよい
```

```
}

class C extends {
  val foo = "Hello" // スーパークラスの初期化の前に呼び出される
} with B {
  def printBar(): Unit = println(bar)
}
```

上記の C の printBar を呼び出しても正しく HelloWorld と表示されます。

この事前定義は利用側からの回避方法ですが、この例の場合はトレイト B のほうに問題がある（普通に使うと初期化の問題が発生してしまう）ので、トレイト B のほうを修正したほうがいいかもしれません。

トレイトの初期化問題は継承されるトレイト側で解決したほうが良いことが多いので、この事前定義の機能は実際のコードではあまり見ることはないかもしれません。

第 10 章

型パラメータ（type parameter）（★★★）

クラスの節では触れませんでしたが、クラスは 0 個以上の型をパラメータとして取ることができます。これは、クラスを作る時点では何の型か特定できない場合（たとえば、コレクションクラスの要素の型）を表したい時に役に立ちます。型パラメータを入れたクラス定義の文法は次のようになります

```
class クラス名 [型パラメータ 1, 型パラメータ 2, ..., 型パラメータ N](コンストラクタ引数 1 :コンストラクタ引数 1 の型, コンストラクタ引
{
  0 個以上のフィールドの定義またはメソッド定義
}
```

型パラメータ 1 から型パラメータ N までは好きな名前を付け、クラス定義の中で使うことができます。とりあえず、簡単な例として、1 個の要素を保持して、要素を入れる（put する）か取りだす（get する）操作ができるクラス Cell を定義してみます。Cell の定義は次のようになります。

```
class Cell[T](var value: T) {
  def put(newValue: T): Unit = {
    value = newValue
  }

  def get(): T = value
}
```

これを REPL で使ってみましょう。

```
scala> class Cell[T](var value: T) {
     |   def put(newValue: T): Unit = {
     |     value = newValue
     |   }
     |
     |   def get(): T = value
     | }
defined class Cell

scala> val cell = new Cell[Int](1)
cell: Cell[Int] = Cell@192aaffb
```

```
scala> cell.put(2)

scala> cell.get()
res1: Int = 2

scala> cell.put("something")
<console>:10: error: type mismatch;
 found   : String("something")
 required: Int
              cell.put("something")
                       ^
```

```
scala> val cell = new Cell[Int](1)
cell: Cell[Int] = Cell@676d572d
```

で、型パラメータとして Int 型を与えて、その初期値として 1 を与えています。型パラメータに Int を与えて Cell をインスタンス化したため、REPL では String を put しようとして、コンパイラにエラーとしてはじかれています。Cell は様々な型を与えてインスタンス化したいクラスであるため、クラス定義時には特定の型を与えることができません。そういった場合に、型パラメータは役に立ちます。

次に、もう少し実用的な例をみてみましょう。メソッドから複数の値を返したい、という要求はプログラミングを行う上でよく発生します。そのような場合、型パラメータが無い言語では、

- 片方を返り値として、もう片方を引数を経由して返す
- 複数の返り値専用のクラスを必要になる度に作る

という選択肢しかありませんでした。しかし、前者は引数を返り値に使うという点で邪道ですし、後者の方法は多数の引数を返したい、あるいは解く問題上で意味のある名前の付けられるクラスであれば良いですが、ただ 2 つの値を返したいといった場合には小回りが効かず不便です。こういう場合、型パラメータを 2 つ取る Pair クラスを作ってしまいます。Pair クラスの定義は次のようになります。toString メソッドの定義は後で表示のために使うだけなので気にしないでください。

```
class Pair[T1, T2](val t1: T1, val t2: T2) {
  override def toString(): String = "(" + t1 + "," + t2 + ")"
}
```

このクラス Pair の利用法としては、たとえば割り算の商と余りの両方を返すメソッド divide が挙げられます。divide の定義は次のようになります。

```
def divide(m: Int, n: Int): Pair[Int, Int] = new Pair[Int, Int](m / n, m % n)
```

これらを REPL にまとめて流し込むと次のようになります。

```
scala> class Pair[T1, T2](val t1: T1, val t2: T2) {
     |   override def toString(): String = "(" + t1 + "," + t2 + ")"
     | }
defined class Pair

scala> def divide(m: Int, n: Int): Pair[Int, Int] = new Pair[Int, Int](m / n, m % n)
divide: (m: Int, n: Int)Pair[Int,Int]

scala> divide(7, 3)
res0: Pair[Int,Int] = (2,1)
```

7 割る 3 の商と余りが res0 に入っていることがわかります。なお、ここでは new Pair[Int, Int](m / n, m % n) としましたが、引数の型から型パラメータの型を推測できる場合、省略できます。この場合、Pair のコンストラクタに与える引数は Int と Int なので、new Pair(m / n, m % n) としても同じ意味になります。この Pair は 2 つの異なる型（同じ型でも良い）を返り値として返したい全ての場合に使うことができます。このように、どの型でも同じ処理を行う場合を抽象化できるのが型パラメータの利点です。

ちなみに、この Pair のようなクラスは Scala ではよく使われるため、Tuple1 から Tuple22(Tuple の後の数字は要素数）があらかじめ用意されています。また、インスタンス化する際も、

```
scala> val m = 7
m: Int = 7

scala> val n = 3
n: Int = 3

scala> new Tuple2(m / n, m % n)
res1: (Int, Int) = (2,1)
```

などとしなくても、

```
scala> val m = 7
m: Int = 7

scala> val n = 3
n: Int = 3

scala> (m / n, m % n)
res2: (Int, Int) = (2,1)
```

とすれば良いようになっています。

変位指定（variance）（★★）

この節では、型パラメータに関する性質である反変、共変について学びます。

共変（covariant）（★★）

Scala では、何も指定しなかった型パラメータは通常は非変（invariant）になります。非変というのは、型パラメータを持ったクラス G、型パラメータ T1 と T2 があったとき、T1 = T2 のときにのみ

```
val : G[T1] = G[T2]
```

というような代入が許されるという性質を表します。これは、違う型パラメータを与えたクラスは違う型になることを考えれば自然な性質です。ここであえて非変について言及したのは、Java の組み込み配列クラスは標準で非変ではなく共変であるという設計ミスを犯しているからです。

ここでまだ共変について言及していなかったので、簡単に定義を示しましょう。共変というのは、型パラメータを持ったクラス G、型パラメータ T1 と T2 があったとき、T1 が T2 を継承しているときにのみ、

```
val : G[T2] = G[T1]
```

というような代入が許される性質を表します。Scala では、クラス定義時に

```
class G[+T]
```

のように型パラメータの前に + を付けるとその型パラメータは（あるいはそのクラスは）共変になります。

このままだと定義が抽象的でわかりづらいかもしれないので、具体的な例として配列型を挙げて説明します。配列型は Java では共変なのに対して Scala では非変であるという点において、面白い例です。まずは Java の例です。G = 配列、 T1 = String, T2 = Object として読んでください。

```
Object[] objects = new String[1];
objects[0] = 100;
```

このコード断片は Java のコードとしてはコンパイルを通ります。ぱっと見でも、Object の配列を表す変数に String の配列を渡すことができるのは理にかなっているように思えます。しかし、このコードを実行すると例外 java.lang.ArrayStoreException が発生します。これは、objects に入っているのが実際には String の配列（String のみを要素として持つ）なのに、2 行目で int 型（ボクシング変換されて Integer 型）の値である 100 を渡そうとしていることによります。

一方、Scala では同様のコードの一行目に相当するコードをコンパイルしようとした時点で、次のようなコンパイルエラーが出ます（Any は全ての型のスーパークラスで、AnyRef に加え、AnyVal（値型）の値も格納できます）。

```
scala scala> val arr: Array[Any] = new Array[String](1) <console>:7: error:
type mismatch;  found   : Array[String]  required: Array[Any]
```

このような結果になるのは、Scala では配列は非変だからです。静的型付き言語の型安全性とは、コンパイル時により多く

のプログラミングエラーを捕捉するものであるとするなら、配列の設計は Scala の方が Java より型安全であると言えます。

さて、Scala では型パラメータを共変にした時点で、安全ではない操作はコンパイラがエラーを出してくれるので安心ですが、共変をどのような場合に使えるかを知っておくのは意味があります。たとえば、先ほど作成したクラス `Pair[T1, T2]` について考えてみましょう。`Pair[T1, T2]` は一度インスタンス化したら、変更する操作ができませんから、`ArrayStoreException` のような例外は起こり得ません。実際、`Pair[T1, T2]` は安全に共変にできるクラスで、`class Pair[+T1, +T2]` のようにしても問題が起きません。

```
scala> class Pair[+T1, +T2](val t1: T1, val t2: T2) {
     |   override def toString(): String = "(" + t1 + "," + t2 + ")"
     | }
defined class Pair

scala> val pair: Pair[AnyRef, AnyRef] = new Pair[String, String]("foo", "bar")
pair: Pair[AnyRef,AnyRef] = (foo,bar)
```

ここで、`Pair` は作成時に値を与えたら後は変更できず、したがって `ArrayStoreException` のような例外が発生する余地がないことがわかります。一般的には、一度作成したら変更できない（immutable）などの型パラメータは共変にしても多くの場合問題がありません。

演習問題

次の *immutable* な *Stack* 型の定義（途中）があります。`???`の箇所を埋めて、*Stack* の定義を完成させなさい。なお、`E >: T` は、`E` は `T` の継承元である、という制約を表しています。また、`Nothing` は全ての型のサブクラスであるような型を表現します。`Stack[T]` は共変なので、`Stack[Nothing]` はどんな型の `Stack` 変数にでも格納することができます。

```
trait Stack[+T] {
  def pop: (T, Stack[T])
  def push[E >: T](e: E): Stack[E]
  def isEmpty: Boolean
}

class NonEmptyStack[+T](private val top: T, private val rest: Stack[T]) extends Stack[T] {
  def push[E >: T](e: E): Stack[E] = ???
  def pop: (T, Stack[T]) = ???
  def isEmpty: Boolean = ???
}

case object EmptyStack extends Stack[Nothing] {
  def pop: Nothing = throw new IllegalArgumentException("empty stack")
  def push[E >: Nothing](e: E): Stack[E] = new NonEmptyStack[E](e, this)
  def isEmpty: Boolean = true
}
```

```
object Stack {
  def apply(): Stack[Nothing] = EmptyStack
}
```

```
class NonEmptyStack[+T](private val top: T, private val rest: Stack[T]) extends Stack[T] {
  def push[E >: T](e: E): Stack[E] = new NonEmptyStack[E](e, this)
  def pop: (T, Stack[T]) = (top, rest)
  def isEmpty: Boolean = false
}
```

反変（contravariant）（★）

次は共変とちょうど対になる性質である反変です。簡単に定義を示しましょう。反変というのは、型パラメータを持ったクラス G、型パラメータ T1 と T2 があったとき、T1 が T2 を継承しているときにのみ、

`val : G[T1] = G[T2]`

というような代入が許される性質を表します。Scala では、クラス定義時に

```
class G[-T]
```

のように型パラメータの前に-を付けるとその型パラメータは（あるいはそのクラスは）反変になります。

反変の例として最もわかりやすいものの 1 つが関数の型です。たとえば、型 T1 と T2 があったとき、

```
val x1: T1 => AnyRef = T2 => AnyRef 型の値
x1(T1 型の値)
```

というプログラムの断片が成功するためには、T1 が T2 を継承する必要があります。その逆では駄目です。仮に、T1 = String, T2 = AnyRef として考えてみましょう。

```
val x1: String => AnyRef = AnyRef => AnyRef 型の値
x1(String 型の値)
```

ここで x1 に実際に入っているのは AnyRef => AnyRef 型の値であるため、引数として String 型の値を与えても、AnyRef 型の引数に String 型の値を与えるのと同様であり、問題なく成功します。T1 と T2 が逆で、T1 = AnyRef, T2 = String の場合、String 型の引数に AnyRef 型の値を与えるのと同様になってしまうので、これは x1 へ値を代入する時点でコンパイルエラーになるべきであり、実際にコンパイルエラーになります。

実際に REPL で試してみましょう。

```
scala> val x1: AnyRef => AnyRef = (x: String) => (x:AnyRef)
<console>:7: error: type mismatch;
 found   : String => AnyRef
 required: AnyRef => AnyRef
       val x1: AnyRef => AnyRef = (x: String) => (x:AnyRef)
                                                 ^

scala> val x1: String => AnyRef = (x: AnyRef) => x
x1: String => AnyRef = <function1>
```

このように、先ほど述べたような結果になっています。

型パラメータの境界（bounds）（★★）

型パラメータ T に対して何も指定しない場合、その型パラメータ T は、どんな型でも入り得ることしかわかりません。そのため、何も指定しない型パラメータ T に対して呼び出せるメソッドは Any に対するもののみになります。しかし、たとえば、順序がある要素からなるリストをソートしたい場合など、T に対して制約を書けると便利な場合があります。そのような場合に使えるのが、型パラメータの境界（bounds）です。型パラメータの境界には 2 種類あります。

上限境界（upper bounds）

1 つ目は、型パラメータがどのような型を継承しているかを指定する上限境界（upper bounds）です。上限境界では、型パラメータの後に、<: を記述し、それに続いて制約となる型を記述します。以下では、show によって文字列化できるクラス Show を定義したうえで、Show であるような型のみを要素として持つ ShowablePair を定義しています。

```
abstract class Show {
  def show: String
}
class ShowablePair[T1 <: Show, T2 <: Show](val t1: T1, val t2: T2) extends Show {
  override def show: String = "(" + t1.show + "," + t2.show + ")"
}
```

ここで、型パラメータ T1、T2 ともに上限境界として Show が指定されているため、t1 と t2 に対して show を呼び出すことができます。なお、上限境界を明示的に指定しなかった場合、Any が指定されたものとみなされます。

下限境界（lower bounds）

2 つ目は、型パラメータがどのような型のスーパータイプであるかを指定する下限境界（lower bounds）です。下限境界は、共変パラメータと共に用いることが多い機能です。実際に例を見ます。

まず、共変の練習問題であったような、イミュータブルな Stack クラスを定義します。この Stack

は共変にしたいとします。

```
abstract class Stack[+E]{
  def push(element: E): Stack[E]
  def top: E
  def pop: Stack[E]
  def isEmpty: Boolean
}
```

しかし、この定義は、以下のようなコンパイルエラーになります。

```
error: covariant type E occurs in contravariant position in type E of value element
         def push(element: E): Stack[E]
                           ^
```

このコンパイルエラーは、共変な型パラメータ T が反変な位置（反変な型パラメータが出現できる箇所）に出現したということを言っています。一般に、引数の位置に共変型パラメータ T の値が来た場合、型安全性が壊れる可能性があるため、このようなエラーが出ます。しかし、この Stack は配列と違ってイミュータブルであるため、本来ならば型安全性上の問題は起きません。この問題に対処するために型パラメータの下限境界を使うことができます。型パラメータ F を push に追加し、その下限境界として、Stack の型パラメータ E を指定します。

```
abstract class Stack[+E]{
  def push[F >: E](element: F): Stack[F]
  def top: E
  def pop: Stack[E]
  def isEmpty: Boolean
}
```

このようにすることによって、コンパイラは、Stack には E の任意のスーパータイプの値が入れられる可能性があることがわかるようになります。そして、型パラメータ F は共変ではないため、どこに出現しても構いません。このようにして、下限境界を利用して、型安全な Stack と共変性を両立することができます。

第 11 章

Scala の関数（★★★）

Scala の関数は、他の言語の関数と扱いが異なります。Scala の関数は単に Function0 ‖ Function22 までのトレイトの無名サブクラスのインスタンスなのです。

たとえば、2 つの整数を取って加算した値を返す add 関数は次のようにして定義することができます：

```
scala> val add = new Function2[Int, Int, Int]{
     |   def apply(x: Int, y: Int): Int = x + y
     | }
add: (Int, Int) => Int = <function2>

scala> add.apply(100, 200)
res0: Int = 300

scala> add(100, 200)
res1: Int = 300
```

Function0 から Function22 までの全ての関数は引数の数に応じた apply メソッドを定義する必要があります。apply メソッドは Scala コンパイラから特別扱いされ、x.apply(y) は常に x(y) のように書くことができます。後者の方が関数の呼び方としては自然ですね。

また、関数を定義するといっても、単に Function0 から Function22 までのトレイトの無名サブクラスのインスタンスを作っているだけです。

無名関数（★★★）

前項で Scala で関数を定義しましたが、これを使ってプログラミングをするとコードが冗長になり過ぎます。そのため、Scala では Function0 ‖ Function22 までのトレイトのインスタンスを生成するためのシンタックスシュガーが用意されています。たとえば、先ほどの add 関数は

```
scala> val add = (x: Int, y: Int) => x + y
add: (Int, Int) => Int = <function2>
```

と書くことができます。ここで、add には単に関数オブジェクトが入っているだけであって、関数

本体には何の名前も付いていないことに注意してください。この、add の右辺のような定義を Scala では無名関数と呼びます。無名関数は単なる FunctionN オブジェクトですから、自由に変数や引数に代入したり返り値として返すことができます。このような、関数を自由に変数や引数に代入したり返り値として返すことができる性質を指して、Scala では関数が第一級の値（First Class Object）であるといいます。

無名関数の一般的な構文は次のようになります。

```
(n1: N1, n2: N2, n3: N3, ...nn: NN) => B
```

n1 から nn までが仮引数の定義で N1 から NN までが仮引数の型です。B は無名関数の本体です。無名関数の返り値の型は通常は B の型から推論されます。先ほど述べたように、Scala の関数は Function0 ‖ Function22 までのトレイトの無名サブクラスのインスタンスですから、引数の最大個数は 22 個になります。

関数の型（★★★）

このようにして定義した関数の型は、本来は FunctionN[...] のようにして記述しなければいけませんが、関数の型については特別にシンタックスシュガーが設けられています。一般に、

```
(n1: N1, n2: N2, n3: N3, ...nn: NN) => B
```

となるような関数の型は FunctionN[N1, N2, N3, ...NN, B の型] と書く代わりに

```
(N1, N2, N3, ...NN) => B の型
```

として記述することができます。直接 FunctionN を型として使うことは稀なので、こちらのシンタックスシュガーを覚えておくと良いでしょう。

関数のカリー化（★★）

メソッドの説明のところでメソッドのカリー化について説明しました。Scala ではメソッドのカリー化を特別に言語でサポートしているからです。一方で、関数のカリー化という概念が存在します。たとえば (Int, Int) => Int 型の関数は常に Int => Int => Int という関数で表現できるという一般的な性質で Scala 上では何も特別な扱いをしていません。試しに上記の add をカリー化してみましょう。

```
scala> val add = (x: Int, y: Int) => x + y
add: (Int, Int) => Int = <function2>

scala> val addCurried = (x: Int) => ((y: Int) => x + y)
addCurried: Int => (Int => Int) = <function1>
```

```
scala> add(100, 200)
res2: Int = 300

scala> addCurried(100)(200)
res3: Int = 300
```

無名関数を定義する構文をネストさせて使っているだけで、何も特別なことはしていないことがわかります。Scala のライブラリの中にはカリー化された形式の関数を要求するものがあったりするので、とりあえず技法として覚えておくのが良いでしょう。

メソッドと関数の違い（★★★）

メソッドについては既に説明しましたが、メソッドと関数の違いについては Scala を勉強する際に注意する必要があります。本来は `def` で始まる構文で定義されたものだけがメソッドなのですが、説明の便宜上、所属するオブジェクトの無いメソッド（今回は説明していません）や REPL で定義したメソッドを関数と呼んだりすることがあります。書籍や Web でもこの 2 つを意図的に、あるいは無意識に混同している例が多々あるので（Scala のバイブル『Scala スケーラブルプログラミング』でも意図的なメソッドと関数の混同の例がいくつかあります）注意してください。

再度強調すると、メソッドは `def` で始まる構文で定義されたものであり、それを関数と呼ぶのはあくまで説明の便宜上であるということです。ここまでメソッドと関数の違いについて強調してきましたが、それは、メソッドは第一級の値ではないのに対して関数は第一級の値であるという大きな違いがあるからです。メソッドを取る引数やメソッドを返す関数、メソッドが入った変数といったものは Scala には存在しません。

高階関数（★★★）

関数を引数に取ったり関数を返すメソッドや関数のことを高階関数と呼びます。先ほどメソッドと関数の違いについて説明したばかりなのに、メソッドのことも関数というのはいささか奇妙ですが、慣習的にそう呼ぶものだと思ってください。

早速高階関数の例についてみてみましょう。

```
scala> def double(n: Int, f: Int => Int): Int = {
     |   f(f(n))
     | }
double: (n: Int, f: Int => Int)Int
```

これは与えられた関数 `f` を 2 回 `n` に適用する関数 `double` です。ちなみに、高階関数に渡される関数は適切な名前が付けられないことも多く、その場合は `f` や `g` などの 1 文字の名前をよく使います。他の関数型プログラミング言語でも同様の慣習があります。呼び出しは次のようになります。

```
scala> double(1, m => m * 2)
res4: Int = 4

scala> double(2, m => m * 3)
res5: Int = 18

scala> double(3, m => m * 4)
res6: Int = 48
```

最初の呼び出しは 1 に対して、与えられた引数を 2 倍する関数を渡していますから、`1 * 2 * 2 = 4` になります。2 番めの呼び出しは 2 に対して、与えられた引数を 3 倍する関数を渡していますから、`2 * 3 * 3 = 18` になります。最後の呼び出しは、3 に対して与えられた引数を 4 倍する関数を渡していますから、`3 * 4 * 4 = 48` になります。

もう少し意味のある例を出してみましょう。プログラムを書くとき、

1. 初期化
2. 何らかの処理
3. 後始末処理

というパターンは頻出します。これをメソッドにした高階関数 `around` を定義します。

```
scala> def around(init: () => Unit, body: () => Any, fin: () => Unit): Any = {
     |   init()
     |   try {
     |     body()
     |   } finally {
     |     fin()
     |   }
     | }
around: (init: () => Unit, body: () => Any, fin: () => Unit)Any
```

`try-finally` 構文は、後の例外処理の節でも出てきますが、大体 Java のそれと同じだと思ってください。この `around` 関数は次のようにして使うことができます。

```
scala> around(
     |   () => println("ファイルを開く"),
     |   () => println("ファイルに対する処理"),
     |   () => println("ファイルを閉じる")
     | )
ファイルを開く
ファイルに対する処理
ファイルを閉じる
res7: Any = ()
```

`around` に渡した関数が順番に呼ばれていることがわかります。ここで、`body` の部分で例外を発生させてみます。`throw` は Java のそれと同じで例外を投げるための構文です。

```
scala> around(
     |   () => println("ファイルを開く"),
     |   () => throw new Exception("例外発生！ "),
     |   () => println("ファイルを閉じる")
     | )
ファイルを開く
ファイルを閉じる
java.lang.Exception: 例外発生！
  at $anonfun$3.apply(<console>:16)
  at $anonfun$3.apply(<console>:16)
  at .around(<console>:15)
  ... 806 elided
```

body の部分で例外が発生しているにも関わらず、fin の部分はちゃんと実行されていることがわかります。ここで、around という高階関数を定義することで、

1. 初期化
2. 何らかの処理
3. 後始末処理

のそれぞれを部品化して、「何らかの処理」の部分で異常が発生しても必ず後始末処理を実行できています。この around メソッドは 1 ‖ 3 の手順を踏む様々な処理に流用することができます。一方、1 ‖ 3 のそれぞれは呼び出し側で自由に与えることができます。このように処理を値として部品化することは高階関数を定義する大きなメリットの 1 つです。

ちなみに、Java 7 では後始末処理を自動化する try-with-resources 文が言語として取り入れられましたが、高階関数のある言語では、言語に頼らず自分でそのような働きをするメソッドを定義することができます。

後のコレクションの節を読むことで、高階関数のメリットをより具体的に理解できるようになるでしょう。

第 12 章

Scala のコレクションライブラリ（immutable と mutable）

Scala には配列（Array）やリスト（List）、連想配列（Map）、集合（Set）を扱うための豊富なライブラリがあります。これを使いこなすことで、Scala でのプログラミングは劇的に楽になります。注意しなければならないのは、Scala では一度作成したら変更できない（immutable）なコレクションと変更できる通常のコレクション（mutable）があることです。皆さんは mutable なコレクションに馴染んでいるかと思いますが、Scala で関数型プログラミングを行うためには、immutable なコレクションを活用する必要があります。

immutable なコレクションを使うのにはいくつものメリットがあります

- 関数型プログラミングで多用する再帰との相性が良い
- 高階関数を用いて簡潔なプログラムを書くことができる
- 一度作ったコレクションが知らない箇所で変更されていない事を保証できる
- 並行に動作するプログラムの中で、安全に受け渡しすることができる

mutable なコレクションを効果的に使えばプログラムの実行速度を上げることができますが、mutable なコレクションをどのような場面で使えばいいかは難しい問題です。

この節では、Scala のコレクションライブラリに含まれる以下のものについての概要を説明します。

- Array(mutable)
- List(immutable)
- Map(immutable)・Map(mutable)
- Set(immutable)・ Set(mutable)

Array（★★）

まずは大抵のプログラミング言語にある配列です。

```
scala> val arr = Array(1, 2, 3, 4, 5)
arr: Array[Int] = Array(1, 2, 3, 4, 5)
```

これで 1 から 5 までの要素を持った配列が arr に代入されました。Scala の配列は、他の言語のそれと同じように要素の中身を入れ替えることができます。配列の添字は 0 から始まります。なお、配列の型を指定しなくて良いのは、Array(1, 2, 3, 4, 5) の部分で、要素型が Int であるに違いないとコンパイラが型推論してくれるからです。型を省略せずに書くと

```
scala> val arr = Array[Int](1, 2, 3, 4, 5)
arr: Array[Int] = Array(1, 2, 3, 4, 5)
```

となります。ここで、[Int] の部分は型パラメータと呼びます。Array だけだとどの型かわからないので、[Int] を付けることでどの型の Array かを指定しているわけです。この型パラメータは型推論を補うために、色々な箇所で出てくるので覚えておいてください。しかし、この場面では、Array の要素型は Int だとわかっているので、冗長です。次に要素へのアクセスと代入です。

```
scala> arr(0) = 7

scala> arr
res1: Array[Int] = Array(7, 2, 3, 4, 5)

scala> arr(0)
res2: Int = 7
```

他の言語だと arr[0] のようにしてアクセスすることが多いので最初は戸惑うかもしれませんが、慣れてください。配列の 0 番目の要素がちゃんと 7 に入れ替わっていますね。

配列の長さは arr.length で取得することができます。

```
scala> arr.length
res3: Int = 5
```

Array[Int] は Java では int[] と同じ意味です。Scala では、配列などのコレクションの要素型を表記するとき Collection[ElementType] のように一律に表記し、配列も同じように記述するのです。Java では配列型だけ特別扱いするのに比べると統一的だと言えるでしょう。

ただし、あくまでも表記上はある程度統一的に扱えますが、実装上は JVM の配列であり、**要素が同じでも equals の結果が true にならない, 生成する際に ClassTag というものが必要**などのいくつかの罠があるので、Array はパフォーマンス上必要になる場合以外はあまり積極的に使うものではありません。

練習問題

配列の i 番目の要素と j 番目の要素を入れ替える swapArray メソッドを定義してみましょう。swapArray メソッドの宣言は

```
def swapArray[T](arr: Array[T])(i: Int, j: Int): Unit = ???
```

となります。i と j が配列の範囲外である場合は特に考慮しなくて良いです。

```
def swapArray[T](arr: Array[T])(i: Int, j: Int): Unit = {
  val tmp = arr(i)
  arr(i) = arr(j)
  arr(j) = tmp
}
```

```
scala> val arr = Array(1, 2, 3, 4, 5)
arr: Array[Int] = Array(1, 2, 3, 4, 5)

scala> swapArray(arr)(0, 4)

scala> arr
res5: Array[Int] = Array(5, 2, 3, 4, 1)

scala> swapArray(arr)(1, 3)

scala> arr
res7: Array[Int] = Array(5, 4, 3, 2, 1)
```

Range（★★）

Range は範囲を表すオブジェクトです。Range は直接名前を指定して生成するより、to メソッドと until メソッドを用いて呼びだすことが多いです。また、toList メソッドを用いて、その範囲の数値の列を後述する List に変換することができます。では、早速 REPL で Range を使ってみましょう。

```
scala> 1 to 5
res8: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5)

scala> (1 to 5).toList
res9: List[Int] = List(1, 2, 3, 4, 5)

scala> 1 until 5
res10: scala.collection.immutable.Range = Range(1, 2, 3, 4)

scala> (1 until 5).toList
res11: List[Int] = List(1, 2, 3, 4)
```

to は右の被演算子を含む範囲を、until は右の被演算子を含まない範囲を表していることがわかります。また、Range は toList で後述する List に変換することができることもわかります。

List（★★★）

さて、導入として大抵の言語にある Array を出しましたが、Scala では Array を使うことはそれほど多くありません。代わりに List や Vector といったデータ構造をよく使います（Vector については後述します）。List の特徴は、一度作成したら中身を変更できない（immutable）ということです。中身を変更できないデータ構造（永続データ構造とも呼びます）は Scala がサポートしている関数型プログラミングにとって重要な要素です。それでは List を使ってみましょう。

```
scala> val lst = List(1, 2, 3, 4, 5)
lst: List[Int] = List(1, 2, 3, 4, 5)
```

```
scala> lst(0) = 7
<console>:14: error: value update is not a member of List[Int]
       lst(0) = 7
       ^
```

見ればわかるように、List は一度作成したら値を更新することができません。しかし、List は値を更新することができませんが、ある List を元に新しい List を作ることができます。これが値を更新することの代わりになります。以降、List に対して組み込みで用意されている各種操作をみていくことで、List の値を更新することなく色々な操作ができることがわかるでしょう。

Nil：空の List

まず最初に紹介するのは Nil です。Scala で空の List を表すには Nil というものを使います。Ruby などでは nil は言語上かなり特別な意味を持ちますが、Scala ではデフォルトでスコープに入っているということ以外は特別な意味はなく単に object です。Nil は単体では意味がありませんが、次に説明する::と合わせて用いることが多いです。

:: - List の先頭に要素をくっつける

::（コンスと読みます）は既にある List の先頭に要素をくっつけるメソッドです。これについては、REPL で結果をみた方が早いでしょう。

```
scala> val a1 = 1 :: Nil
a1: List[Int] = List(1)

scala> val a2 = 2 :: a1
a2: List[Int] = List(2, 1)

scala> val a3 = 3 :: a2
a3: List[Int] = List(3, 2, 1)
```

```
scala> val a4 = 4 :: a3
a4: List[Int] = List(4, 3, 2, 1)

scala> val a5 = 5 :: a3
a5: List[Int] = List(5, 3, 2, 1)
```

付け足したい要素を::を挟んで List の前に書くことで List の先頭に要素がくっついていることがわかります。ここで、::はやや特別な呼び出し方をするメソッドであることを説明しなければなりません。まず、Scala では 1 引数のメソッドは中置記法で書くことができます。それで、1 :: Nil のように書くことができるわけです。次に、メソッド名の最後が:で終わる場合、被演算子の前と後ろをひっくり返して右結合で呼び出します。たとえば、

```
scala> 1 :: 2 :: 3 :: 4 :: Nil
res13: List[Int] = List(1, 2, 3, 4)
```

は、実際には、

```
scala> Nil.::(4).::(3).::(2).::(1)
res14: List[Int] = List(1, 2, 3, 4)
```

のように解釈されます。List の要素が演算子の前に来て、一見数値のメソッドのように見えるのに List のメソッドとして呼び出せるのはそのためです。

++：List 同士の連結

++ は List 同士を連結するメソッドです。これも REPL で見た方が早いでしょう。

```
scala> List(1, 2) ++ List(3, 4)
res15: List[Int] = List(1, 2, 3, 4)

scala> List(1) ++ List(3, 4, 5)
res16: List[Int] = List(1, 3, 4, 5)

scala> List(3, 4, 5) ++ List(1)
res17: List[Int] = List(3, 4, 5, 1)
```

++ は 1 引数のメソッドなので、中置記法で書いています。また、末尾が:で終わっていないので、たとえば、

```
scala> List(1, 2) ++ List(3, 4)
res18: List[Int] = List(1, 2, 3, 4)
```

は

```
scala> List(1, 2).++(List(3, 4))
res19: List[Int] = List(1, 2, 3, 4)
```

と同じ意味です。大きな List 同士を連結する場合、計算量が大きくなるのでその点には注意した

方が良いです。

mkString：文字列のフォーマッティング（★★★）

このメソッドは Scala で非常に頻繁に使用され皆さんも、Scala を使っていく上で使う機会が多いであろうメソッドです。このメソッドは引数によって多重定義されており、3 バージョンあるのでそれぞれを紹介します。

mkString

引数なしバージョンです。このメソッドは、単に List の各要素を左から順に繋げた文字列を返します。

```
scala> List(1, 2, 3, 4, 5).mkString
res20: String = 12345
```

注意しなければならないのは、引数なしメソッドの mkString は () を付けて呼びだすことができないという点です。たとえば、以下のコードは、若干分かりにくいエラーメッセージがでてコンパイルに失敗します。

```
scala> List(1, 2, 3, 4, 5).mkString()
<console>:13: error: overloaded method value mkString with alternatives:
  => String <and>
  (sep: String)String <and>
  (start: String,sep: String,end: String)String
 cannot be applied to ()
       List(1, 2, 3, 4, 5).mkString()
                           ^
```

Scala の 0 引数メソッドは () なしと () を使った定義の二通りあって、前者の形式で定義されたメソッドは () を付けずに呼び出さなければいけません。逆に、() を使って定義されたメソッドは、() を付けても付けなくても良いことになっています。この Scala の仕様は混乱しやすいので注意してください。

mkString(sep: String)

引数にセパレータ文字列 sep を取り、List の各要素を sep で区切って左から順に繋げた文字列を返します。

```
scala> List(1, 2, 3, 4, 5).mkString(",")
res22: String = 1,2,3,4,5
```

`mkString(start: String, sep: String, end: String)`

`mkString(sep)` とほとんど同じですが、`start` と `end` に囲まれた文字列を返すところが異なります。

```
scala> List(1, 2, 3, 4, 5).mkString("[", ",", "]")
res23: String = [1,2,3,4,5]
```

練習問題

`mkString` を使って、最初の数 `start` と最後の数 `end` を受け取って、

`start,...,end`

となるような文字列を返すメソッド `joinByComma` を定義してみましょう（ヒント：`Range` にも `mkString` メソッドはあります）。

```
def joinByComma(start: Int, end: Int): String = {
  ???
}
```

```
def joinByComma(start: Int, end: Int): String = {
  (start to end).mkString(",")
}
```

```
scala> joinByComma(1, 10)
res24: String = 1,2,3,4,5,6,7,8,9,10
```

foldLeft：左からの畳み込み（★★★）

`foldLeft` メソッドは `List` にとって非常に基本的なメソッドです。他の様々なメソッドを `foldLeft` を使って実装することができます。`foldLeft` の宣言を Scala の API ドキュメントから引用すると、

```
def foldLeft[B](z: B)(f: (B, A) ⇒ B): B
```

となります。`z` が `foldLeft` の結果の初期値で、リストを左からたどりながら `f` を適用していきます。`foldLeft` についてはイメージが湧きにくいと思いますので、`List(1, 2, 3).foldLeft(0)((x, y) => x + y)` の結果を図示します。

```
    +   3
   / \
  +   2
 / \
0   1
```

この図で、

- / \ 0 1 “‘

は + に 0 と 1 を与えて適用するということを意味します。リストの要素を左から順に f を使って「畳み込む」（fold は英語で畳み込むという意味を持ちます）状態がイメージできるでしょうか。foldLeft は汎用性の高いメソッドで、たとえば、List の要素の合計を求めたい場合は

```
scala> List(1, 2, 3).foldLeft(0)((x, y) => x + y)
res25: Int = 6
```

List の要素を全て掛けあわせた結果を求めたい場合は

```
scala> List(1, 2, 3).foldLeft(1)((x, y) => x * y)
res26: Int = 6
```

とすることで求める結果を得ることができます[*1]。その他にも様々な処理を foldLeft を用いて実装することができます。

練習問題

foldLeft を用いて、List の要素を反転させるメソッド reverse を実装してみましょう。

```
def reverse[T](list: List[T]): List[T] = list.foldLeft(Nil: List[T])((a, b) => b :: a)
```

```
scala> reverse(List(1, 2, 3, 4, 5))
res27: List[Int] = List(5, 4, 3, 2, 1)
```

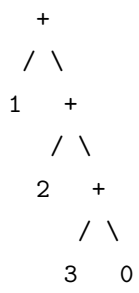
foldRight：右からの畳み込み（★★）

foldLeft が List の左からの畳み込みだったのに対して、foldRight は右からの畳込みです。foldRight の宣言を Scala の API ドキュメントから参照すると、

[*1] ただし、これはあくまでも foldLeft の例であって、要素の和や積を求めたい場合に限って言えばもっと便利なメソッドが標準ライブラリに存在するので、実際にはこの例のような使い方はしません

```
def foldRight[B](z: B)(op: (A, B) ⇒ B): B
```

となります。foldRight に与える関数である op の引数の順序が foldLeft の場合と逆になっている事に注意してください。foldRight を List(1, 2, 3).foldRight(0)((y, x) => y + x) とした場合の様子を図示すると次のようになります。



ちょうど foldLeft と対称になっています。foldRight も非常に汎用性の高いメソッドで、多くの処理を foldRight を用いて実装することができます。

練習問題

List の全ての要素を足し合わせるメソッド sum を foldRight を用いて実装してみましょう。sum の宣言は次のようになります。なお、List が空のときは 0 を返してみましょう。

```
def sum(list: List[Int]): Int = ???
```

```
def sum(list: List[Int]): Int = list.foldRight(0){(x, y) => x + y}
```

```
scala> sum(List(1, 2, 3, 4, 5))
res28: Int = 15
```

練習問題

List の全ての要素を掛け合わせるメソッド mul を foldRight を用いて実装してみましょう。mul の宣言は次のようになります。なお、List が空のときは 1 を返してみましょう。

```
scala> def mul(list: List[Int]): Int = ???
mul: (list: List[Int])Int
```

```
def mul(list: List[Int]): Int = list.foldRight(1){(x, y) => x * y}
```

```
scala> mul(List(1, 2, 3, 4, 5))
res29: Int = 120
```

練習問題

mkString を実装してみましょう。mkString そのものを使ってはいけませんが、foldLeft や foldRight などの List に定義されている他のメソッドは自由に使って構いません。List の API リファレンスを読めば必要なメソッドが載っています。実装する mkString の宣言は

```
def mkString[T](list: List[T])(sep: String): String = ???
```

となります。残りの 2 つのバージョンの mkString は実装しなくても構いません。

```
def mkString[T](list: List[T])(sep: String): String = list match {
  case Nil => ""
  case x::xs => xs.foldLeft(x.toString){ case (x, y) => x + sep + y }
}
```

map：各要素を加工した新しい List を返す（★★★）

map メソッドは、1 引数の関数を引数に取り、各要素に関数を適用した結果できた要素からなる新たな List を返します。ためしに List(1, 2, 3, 4, 5) の各要素を 2 倍してみましょう。

```
scala> List(1, 2, 3, 4, 5).map(x => x * 2)
res30: List[Int] = List(2, 4, 6, 8, 10)
```

x => x * 2 の部分は既に述べたように、無名関数を定義するための構文です。メソッドの引数に与える短い関数を定義するときは、Scala では無名関数をよく使います。List の全ての要素に何らかの処理を行い、その結果を加工するという処理は頻出するため、map は Scala のコレクションのメソッドの中でも非常によく使われるものになっています。

練習問題

map メソッドを foldLeft と reverse を使って実装してみましょう。

```
def map[T, U](list: List[T])(f: T => U): List[U] = {
  list.foldLeft(Nil:List[U]){(x, y) => f(y) :: x}.reverse
}
```

filter：条件に合った要素だけを抽出した新しい List を返す（★★★）

filter メソッドは、Boolean 型を返す 1 引数の関数を引数に取り、各要素に関数を適用し、true になった要素のみを抽出した新たな List を返します。List(1, 2, 3, 4, 5) から奇数だけを抽出

してみましょう。

```
scala> List(1, 2, 3, 4, 5).filter(x => x % 2 == 1)
res31: List[Int] = List(1, 3, 5)
```

練習問題

filter メソッドを foldLeft と reverse を使って実装してみましょう。

scala def filter[T](list: List[T])(f: T => Boolean): List[T] = { list.foldLeft(Nil:List[T]){(y) => if(f(y)) y::x else x}.reverse }

find：条件に合った最初の要素を返す（★★★）

find メソッドは、Boolean 型を返す 1 引数の関数を引数に取り、各要素に前から順番に関数を適用し、最初に true になった要素を Some でくるんだ値を Option 型として返します。1 つの要素もマッチしなかった場合 None を Option 型として返します。List(1, 2, 3, 4, 5) から最初の奇数だけを抽出してみましょう

```
scala> List(1, 2, 3, 4, 5).find(x => x % 2 == 1)
res32: Option[Int] = Some(1)
```

後で説明されることになりますが、Option 型は Scala プログラミングの中で重要な要素であり頻出します。

takeWhile：先頭から条件を満たしている間を抽出する（★★）

takeWhile メソッドは、Boolean 型を返す 1 引数の関数を引数に取り、前から順番に関数を適用し、結果が true の間のみからなる List を返します。List(1, 2, 3, 4, 5) の 5 より前の 4 要素を抽出してみます。

```
scala> List(1, 2, 3, 4, 5).takeWhile(x => x != 5)
res33: List[Int] = List(1, 2, 3, 4)
```

count：List の中で条件を満たしている要素の数を計算する（★★）

count メソッドは、Boolean 型を返す 1 引数の関数を引数に取り、全ての要素に関数を適用して、true が返ってきた要素の数を計算します。例として List(1, 2, 3, 4, 5) の中から偶数の数（2 になるはず）を計算してみます。

```
scala> List(1, 2, 3, 4, 5).count(x => x % 2 == 0)
res34: Int = 2
```

練習問題

count メソッドを foldLeft を使って実装してみましょう。

```
def count(list: List[Int])(f: Int => Boolean): Int  = {
  list.foldLeft(0){(x, y) => if(f(y)) x + 1 else x}
}
```

flatMap：List をたいらにする（★★★）

flatMap は一見少し変わったメソッドですが、後々重要になってくるメソッドなので説明しておきます。flatMap の宣言は Scala の API ドキュメントから参照すると、

```
final def flatMap[B](f: (A) ⇒ GenTraversableOnce[B]): List[B]
```

となります。ここで、GenTraversableOnce[B] という変わった型が出てきていますが、ここではあらゆるコレクションを入れることができる型程度に考えてください。さて、flatMap の引数 f の型は (A) => GenTraversableOnce[B] です。flatMap はこれを使って、各要素に f を適用して、結果の要素からなるコレクションを分解して List の要素にします。これについては、実際に見た方が早いでしょう。

```
scala> List(List(1, 2, 3), List(4, 5)).flatMap{e => e.map{g => g + 1}}
res35: List[Int] = List(2, 3, 4, 5, 6)
```

ネストした List の各要素に flatMap の中で map を適用して、List の各要素に 1 を足したものをたいらにしています。これだけだとありがたみがわかりにくいですが、ちょっと形を変えてみると非常に面白い使い方ができます：

```
scala> List(1, 2, 3).flatMap{e => List(4, 5).map(g => e * g)}
res36: List[Int] = List(4, 5, 8, 10, 12, 15)
```

List(1, 2, 3) と List(4, 5) の 2 つの List についてループし、各々の要素を掛けあわせた要素からなる List を抽出しています。実は、for-comprehension

```
for(x <- col1; y <- col2;) yield z
```

は

```
col1.flatMap{x => col2.map{y => z}}
```

のシンタックスシュガーだったのです。すなわち、ある自分で定義したデータ型に flatMap と map を（適切に）実装すれば for 構文の中で使うことができるのです。

List の性能特性（★★★）

List の性能特性として、List の先頭要素へのアクセスは高速にできる反面、要素へのランダムアクセスや末尾へのデータの追加は List の長さに比例した時間がかかってしまうということが挙げられます。List は関数型プログラミング言語で最も基本的なデータ構造で、どの関数型プログラミング言語でもたいていは List がありますが、その性能特性には十分注意して扱う必要があります。特に他の言語のプログラマはうっかり List の末尾に要素を追加するような遅いプログラムを書いてしまうことがあるので注意する必要があります。

```
scala> List(1, 2, 3, 4)
res37: List[Int] = List(1, 2, 3, 4)

scala> 5 :: List(1, 2, 3, 4) // List の先頭のセルに新しいをくっつける
res38: List[Int] = List(5, 1, 2, 3, 4)

scala> List(1, 2, 3, 4) :+ 5 // 注意！ 末尾への追加は、List の要素数分かかる
res39: List[Int] = List(1, 2, 3, 4, 5)
```

紹介したメソッドについて（★）

mkString をはじめとした List の色々なメソッドを紹介してきましたが、実はこれらの大半は List 特有ではなく、既に紹介した Range や Array、これから紹介する他のコレクションでも同様に使うことができます。何故ならばこれらの操作の大半は特定のコレクションではなく、コレクションのスーパータイプである共通のトレイト中に宣言されているからです。もちろん、List に要素を加える処理と Set に要素を加える処理（Set に既にある要素は加えない）のように、中で行われる処理が異なることがあるので、その点は注意する必要があります。詳しくは Scala の API ドキュメントを探索してみましょう。

Vector（★★★）

Vector は少々変わったデータ構造です。Vector は一度データ構造を構築したら変更できない immutable なデータ構造です。要素へのランダムアクセスや長さの取得、データの挿入や削除、いずれの操作も十分に高速にできる比較的万能なデータ構造です。immutable なデータ構造を使う場合は、まず Vector を検討すると良いでしょう。

```
scala> Vector(1, 2, 3, 4, 5) //どの操作も「ほぼ」一定の時間で終わる
res40: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3, 4, 5)

scala> 6 +: Vector(1, 2, 3, 4, 5)
res41: scala.collection.immutable.Vector[Int] = Vector(6, 1, 2, 3, 4, 5)

scala> Vector(1, 2, 3, 4, 5) :+ 6
```

```
res42: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3, 4, 5, 6)

scala> Vector(1, 2, 3, 4, 5).updated(2, 5)
res43: scala.collection.immutable.Vector[Int] = Vector(1, 2, 5, 4, 5)
```

Map（★★★）

Map はキーから値へのマッピングを提供するデータ構造です。他の言語では辞書や連想配列と呼ばれたりします。Scala では Map として一度作成したら変更できない immutable な Map と変更可能な mutable な Map の 2 種類を提供しています。

scala.collection.immutable.Map

Scala で何も設定せずにただ Map と書いた場合、scala.collection.immutable.Map が使われます。その名の通り、一度作成すると変更することはできません。内部の実装としては主に scala.collection.immutable.HashMap と scala.collection.immutable.TreeMap の 2 種類がありますが、通常は HashMap が使われます。

```
scala> val m = Map("A" -> 1, "B" -> 2, "C" -> 3)
m: scala.collection.immutable.Map[String,Int] = Map(A -> 1, B -> 2, C -> 3)

scala> m.updated("B", 4) //一見元の Map を変更したように見えても
res44: scala.collection.immutable.Map[String,Int] = Map(A -> 1, B -> 4, C -> 3)

scala> m // 元の Map はそのまま
res45: scala.collection.immutable.Map[String,Int] = Map(A -> 1, B -> 2, C -> 3)
```

scala.collection.mutable.Map

Scala の変更可能な Map は scala.collection.mutable.Map にあります。実装としては、scala.collection.mutable.HashMap、scala.collection.mutable.LinkedHashMap、リストをベースにした scala.collection.mutable.ListMap がありますが、通常は HashMap が使われます。

```
scala> import scala.collection.mutable
import scala.collection.mutable

scala> val m = mutable.Map("A" -> 1, "B" -> 2, "C" -> 3)
m: scala.collection.mutable.Map[String,Int] = Map(A -> 1, C -> 3, B -> 2)

scala> m("B") = 5 // B -> 5 のマッピングに置き換える

scala> m // 変更が反映されている
res47: scala.collection.mutable.Map[String,Int] = Map(A -> 1, C -> 3, B -> 5)
```

Set（★）

Set は値の集合を提供するデータ構造です。Set の中では同じ値が 2 つ以上存在しません。たとえば、Int の Set の中には 1 が 2 つ以上含まれていてはいけません。REPL で Set を作成するための式を入力すると、

```
scala> Set(1, 1, 2, 3, 4)
res48: scala.collection.immutable.Set[Int] = Set(1, 2, 3, 4)
```

重複した 1 が削除されて、1 が 1 つだけになっていることがわかります。

scala.collection.immutable.Set

Scala で何も設定せずにただ Set と書いた場合、scala.collection.immutable.Set が使われます。immutable な Map の場合と同じく、一度作成すると変更することはできません。内部の実装としては、主に scala.collection.immutable.HashSet と scala.collection.immutable.TreeSet の 2 種類がありますが、通常は HashSet が使われます。

```
scala> val s = Set(1, 2, 3, 4, 5)
s: scala.collection.immutable.Set[Int] = Set(5, 1, 2, 3, 4)

scala> s - 5 // 5 を削除した後も
res49: scala.collection.immutable.Set[Int] = Set(1, 2, 3, 4)

scala> s // 元の Set はそのまま
res50: scala.collection.immutable.Set[Int] = Set(5, 1, 2, 3, 4)
```

scala.collection.mutable.Set

Scala の変更可能な Set は scala.collection.mutable.Set にあります。主な実装としては、scala.collection.mutable.HashSet、scala.collection.mutable.TreeSet がありますが、通常は HashSet が使われます。

```
scala> import scala.collection.mutable
import scala.collection.mutable

scala> val s = mutable.Set(1, 2, 3, 4, 5)
s: scala.collection.mutable.Set[Int] = Set(1, 5, 2, 3, 4)

scala> s -= 5 // 5 を削除したら
res51: s.type = Set(1, 2, 3, 4)

scala> s // 変更が反映される
res52: scala.collection.mutable.Set[Int] = Set(1, 2, 3, 4)
```

その他資料（★）

さらにコレクションライブラリについて詳しく知りたい場合は、以下の公式のドキュメントなどを読みましょう http://docs.scala-lang.org/ja/overviews/collections/introduction.html

第 13 章

ケースクラスとパターンマッチング（★★★）

パターンマッチングは、Scala を初めとする関数型言語に一般的な機能です。C や Java の switch 文に似ていますが、より強力な機能です。しかし、パターンマッチングの真価を発揮するには、標準ライブラリまたはユーザが定義したケースクラス（case class）によるデータ型の定義が必要になります。

簡単なケースクラスによるデータ型を定義してみます。

```
sealed abstract class DayOfWeek
case object Sunday extends DayOfWeek
case object Monday extends DayOfWeek
case object Tuesday extends DayOfWeek
case object Wednesday extends DayOfWeek
case object Thursday extends DayOfWeek
case object Friday extends DayOfWeek
case object Saturday extends DayOfWeek
```

これは、一週間の曜日を表すデータ型です。C や Java の enum に似ていますね。実際、同じように使うことができます。たとえば、以下のように DayOfWeek 型の変数に Sunday を代入することができます。

```
val x: DayOfWeek = Sunday
```

object またはその他のデータ型は、パターンマッチングのパターンを使うことができます。この例では、この DayOfWeek 型を継承した各 object をパターンマッチングのパターンを使うことができます。パターンマッチングの構文は再度書くと、

```
式 match {
  case pat1 =>
  case pat2 =>
  ...
}
```

のようになります。DayOfWeek の場合、次のようにして使うことができます。

```
scala> x match {
     |   case Sunday => 1
     |   case Monday => 2
     |   case Tuesday => 3
     |   case Wednesday => 4
     |   case Thursday => 5
     |   case Friday => 6
     | }
<console>:21: warning: match may not be exhaustive.
It would fail on the following input: Saturday
       x match {
       ^
res0: Int = 1
```

これは、x が Sunday なら 1 を、Monday なら 2 を…返すパターンマッチです。ここで、パターンマッチで漏れがあった場合、コンパイラが警告してくれます。この警告は、sealed 修飾子をスーパークラス/トレイトに付けることによって、その（直接の）サブクラス/トレイトは同じファイル内にしか定義できないという性質を利用して実現されています。この用途以外で sealed はめったに使われないので、ケースクラスのスーパークラス/トレイトには sealed を付けるものだと覚えておけば良いでしょう。

これだけだと、C や Java の列挙型とあまり変わらないように見えますが、それらと異なるのは各々のデータは独立してパラメータを持つことができることです。また、パターンマッチの際はそのデータ型の種類によって分岐するだけでなく、データを分解することができることが特徴です。

例として四則演算を表す構文木を考えてみます。各ノード Exp を継承し（つまり、全てのノードは式である）、二項演算を表すノードはそれぞれの子として lhs（左辺）、rhs（右辺）を持つこととします。葉ノードとして整数リテラル（Lit）も入れます。これは Int の値を取るものとします。また、二項演算の結果として小数が現れた場合は小数部を切り捨てることとします。これを表すデータ型を Scala で定義すると次のようになります。

```
sealed abstract class Exp
case class Add(lhs: Exp, rhs: Exp) extends Exp
case class Sub(lhs: Exp, rhs: Exp) extends Exp
case class Mul(lhs: Exp, rhs: Exp) extends Exp
case class Div(lhs: Exp, rhs: Exp) extends Exp
case class Lit(value: Int) extends Exp
```

全てのデータ型に case 修飾子がついているので、これらのデータ型はパターンマッチングのパターンとして使うことができます。この定義から、1 + ((2 * 3) / 2) という式を表すノードを構築します。

```
scala> val example = Add(Lit(1), Div(Mul(Lit(2), Lit(3)), Lit(2)))
example: Add = Add(Lit(1),Div(Mul(Lit(2),Lit(3)),Lit(2)))
```

この example ノードを元に四則演算を定義する関数を定義してみます。関数の定義の詳細は後ほ

ど説明しますが、ここでは雰囲気だけをつかんでください。

```
scala> def eval(exp: Exp): Int = exp match {
     |   case Add(l, r) => eval(l) + eval(r)
     |   case Sub(l, r) => eval(l) - eval(r)
     |   case Mul(l, r) => eval(l) * eval(r)
     |   case Div(l, r) => eval(l) / eval(r)
     |   case Lit(v) => v
     | }
eval: (exp: Exp)Int
```

この定義を REPL に読み込ませて、`eval(example)` として、

```
scala> eval(example)
res1: Int = 4
```

のように表示されれば成功です。きちんと、`1 + ((2 * 3) / 2)` という式の計算結果が出ていますね。ここで注目すべきは、パターンマッチングによって、

1. ノードの種類と構造によって分岐する
2. ネストしたノードを分解する
3. ネストしたノードを分解した結果を変数に束縛する

という 3 つの動作が同時に行えていることです。これがケースクラスを使ったデータ型とパターンマッチングの組み合わせの強力さです。また、この `match` 式の中で、たとえば `case Lit(v) => v` の行を書き忘れた場合、`DayOfWeek` の例と同じように、

```
<console>:16: warning: match may not be exhaustive.
It would fail on the following input: Lit(_)
       def eval(exp: Exp): Int = exp match {
```

記述漏れがあることを指摘してくれますから、ミスを防ぐこともできます。

変数宣言におけるパターンマッチング

`match` 式中のパターンマッチングのみを扱ってきましたが、実は変数宣言でもパターンマッチングを行うことができます。

たとえば、次のようなケースクラス `Point` があったとします。

```
scala> case class Point(x: Int, y: Int)
defined class Point
```

このケースクラス `Point` に対して、

```
scala> val Point(x, y) = Point(10, 20)
x: Int = 10
y: Int = 20
```

とすると、x に 10 が、y に 20 が束縛されます。もしパターンにマッチしなかった場合は、例外 scala.MatchError が発生してしまうので、変数宣言におけるパターンマッチングは、それが必ず成功すると型情報から確信できる場合にのみ使うようにしましょう。

練習問題

DayOfWeek 型を使って、ある日の次の曜日を返すメソッド nextDayOfWeek

```
def nextDayOfWeek(d: DayOfWeek): DayOfWeek = ???
```

をパターンマッチを用いて定義してみましょう。

```
def nextDayOfWeek(d: DayOfWeek): DayOfWeek = d match {
  case Sunday => Monday
  case Monday => Tuesday
  case Tuesday => Wednesday
  case Wednesday => Thursday
  case Thursday => Friday
  case Friday => Saturday
  case Saturday => Sunday
}
```

```
scala> nextDayOfWeek(Sunday)
res2: DayOfWeek = Monday

scala> nextDayOfWeek(Monday)
res3: DayOfWeek = Tuesday

scala> nextDayOfWeek(Saturday)
res4: DayOfWeek = Sunday
```

練習問題

二分木（子の数が最大で 2 つであるような木構造）を表す型 Tree と Branch, Empty を考えます：

```
sealed abstract class Tree
case class Branch(value: Int, left: Tree, right: Tree) extends Tree
case object Empty extends Tree
```

子が 2 つで左の子の値が 2、右の子の値が 3、自分自身の値が 1 の木構造はたとえば次のようにして定義することができます。

```
scala> val tree: Tree = Branch(1, Branch(2, Empty, Empty), Branch(3, Empty, Empty))
tree: Tree = Branch(1,Branch(2,Empty,Empty),Branch(3,Empty,Empty))
```

このような木構造に対して、

1. 最大値を求める max メソッド：
2. 最小値を求める min メソッド：
3. 深さを求める depth メソッド：

```
def max(tree: Tree): Int = ???
def min(tree: Tree): Int = ???
def depth(tree: Tree): Int = ???
```

をそれぞれ定義してみましょう。なお、

```
depth(Empty) == 0
depth(Branch(10, Empty, Empty)) = 1
```

です。

余裕があれば木構造を、

左の子孫の全ての値 <= 自分自身の値 < 右の子孫の全部の値

となるような木構造に変換する sort メソッド：

```
def sort(tree: Tree): Tree = ???
```

を定義してみましょう。なお、sort メソッドは、葉ノードでないノードの個数と値が同じであれば元の構造と同じでなくても良いものとします。

```
object BinaryTree {
  sealed abstract class Tree
  case class Branch(value: Int, left: Tree, right: Tree) extends Tree
  case object Empty extends Tree

  def max(t: Tree): Int = t match {
    case Branch(v1, Branch(v2, Empty, Empty), Branch(v3, Empty, Empty)) =>
      val m = if(v1 <= v2) v2 else v1
      if(m <= v3) v3 else m
    case Branch(v1, Branch(v2, Empty, Empty), Empty) => if(v1 <= v2) v2 else v1
    case Branch(v1, Empty, Branch(v2, Empty, Empty)) => if(v1 <= v2) v2 else v1
    case Branch(v, l, r) =>
      val m1 = max(l)
      val m2 = max(r)
      val m3 = if(m1 <= m2) m2 else m1
      if(v <= m3) m3 else v
    case Empty => throw new RuntimeException
  }


  def min(t: Tree): Int = t match {
    case Branch(v1, Branch(v2, Empty, Empty), Branch(v3, Empty, Empty)) =>
      val m = if(v1 >= v2) v2 else v1
      if(m >= v3) v3 else m
```

```
    case Branch(v1, Branch(v2, Empty, Empty), Empty) => if(v1 >= v2) v2 else v1
    case Branch(v1, Empty, Branch(v2, Empty, Empty)) => if(v1 >= v2) v2 else v1
    case Branch(v, l, r) =>
      val m1 = min(l)
      val m2 = min(r)
      val m3 = if(m1 > m2) m2 else m1
      if(v >= m3) m3 else v
    case Empty => throw new RuntimeException
  }

  def depth(t: Tree): Int = t match {
    case Empty => 0
    case Branch(_, l, r) =>
      val ldepth = depth(l)
      val rdepth = depth(r)
      (if(ldepth < rdepth) rdepth else ldepth) + 1
  }

  def sort(t: Tree): Tree = {
    def fromList(list: List[Int]): Tree = {
      def insert(value: Int, t: Tree): Tree = t match {
        case Empty => Branch(value, Empty, Empty)
        case Branch(v, l, r) =>
          if(value <= v) Branch(v, insert(value, l), r)
          else Branch(v, l, insert(value, r))
      }
      list.foldLeft(Empty:Tree){ case (t, v) => insert(v, t) }
    }
    def toList(tree: Tree): List[Int] = tree match {
      case Empty => Nil
      case Branch(v, l, r) => toList(l) ++ List(v) ++ toList(r)
    }
    fromList(toList(t))
  }

  def find(t: Tree, target: Int): Boolean = t match {
    case Branch(v, l, r) => if(v == target) true else (find(l, target) || find(r, target))
    case Empty => false
  }

  def findBinaryTree(t: Tree, target: Int): Boolean = t match {
    case Branch(v, l, r) => if(v == target) true else (if(target <= v) findBinaryTree(l, target) else findBinaryTree(r
    case Empty => false
  }
}
```

第 14 章

エラー処理（★★★）

ここでは Scala におけるエラー処理の基本を学びます。Scala でのエラー処理は例外を使う方法と、Option や Either や Try などのデータ型を使う方法があります。この 2 つの方法はどちらか一方だけを使うわけではなく、状況に応じて使いわけることになります。

まずは私たちが扱わなければならないエラーとエラー処理の性質について確認しましょう。

エラーとは

プログラムにとって、エラーというものにはどういったものがあるのか考えてみます。

ユーザーからの入力

1 つはユーザーから受け取る不正な入力です。たとえば以下のようなものが考えられます。

- 文字数が長すぎる
- 電話番号に文字列を使うなど、正しいフォーマットではない
- 既に登録されているユーザー名を使おうとしている

など色々な問題が考えられます。また悪意のある攻撃者から攻撃を受けることもあります。

- アクセスを制限しているデータを見ようとしている
- ログインセッションの Cookie を改変する
- 大量にアクセスをおこない、システムを利用不能にしようとする

基本的に外から受け取るデータはすべてエラーの原因となりえるので注意が必要です。

外部サービスのエラー

自分たちのプログラムが利用する外部サービスのエラーも考えられます。

- Twitter や Facebook に投稿しようとしても繋がらない

- iPhone や Android と通信しようとしても回線の都合で切れてしまう
- ユーザーにメールを送信しようとしても失敗する

以上のように外部のサービスを使わなければならないような処理はすべて失敗することを想定したほうがいいでしょう。

内部のエラー

外的な要因だけではなく、内部の要因でエラーが発生することもあります。

- ライブラリのバグや自分たちのバグにより、プログラム全体が終了してしまう
- MySQL や Redis などの内部で利用しているサーバーが終了してしまう
- メモリやディスク容量が足りない
- 処理に非常に大きな時間がかかってしまう

内部のエラーは扱うことが難しい場合が多いですが、起こりうることは念頭に置くべきです。

エラー処理で実現しなければならないこと

以上のようなエラーに対して、私たちが行わなければいけないことを挙げてみます。

例外安全性

エラー処理の中の 1 つの例外処理には「例外安全性」という概念があります。例外が発生してもシステムがダウンしたり、データの不整合などの問題が起きない場合、例外安全と言います。

この概念はエラー処理全般にもあてはまります。私たちが作るプログラムを継続的に動作させたいと考えた場合、ユーザーの入力や外部サービスの問題により、システムダウンやデータの不整合が起きてはなりません。これがエラー処理の第一の目的になります。

強い例外安全性

例外安全性にはさらに強い概念として「強い例外安全性」というものがあります。これは例外が発生した場合、すべての状態が例外発生前に戻らなければならないという制約です。一般的にはこの制約を満たすことは難しいのですが、たとえばユーザーがサービスに課金して、何らかのエラーが生じた場合、確実にエラーを検出し、課金処理を取り消さなければなりません。どのような処理に強い例外安全性が求められるか判断し、どのように実現するかを考える必要があります。

Java におけるエラー処理

Java のエラー処理の方法は Scala にも適用できるものが多いです。ここでは Java のエラー処理の注意点についていくつか復習しましょう。

null を返すことでエラーを表現する場合の注意点

Java では、変数が未初期化である場合や、コレクションライブラリが空なのに要素を取得しようとした場合など、null でエラーを表現することがあります。Java はプリミティブ型以外の参照型はすべて null にすることができます。この性質はエラー値を他に用意する必要がないという点では便利なのですが、しばしば返り値を null かどうかチェックするのを忘れて実行時エラーの NullPointerException（通称：ぬるぽ・NPE）を発生させてしまいます。（「ぬるぽ」と「ガッ」というやりとりをする 2ch の文化の語源でもあります）

参照型がすべて `null` になりうるということは、メソッドが `null` が返されるかどうかはメソッドの型からはわからないので、Java のメソッドで `null` を返す場合はドキュメントに書くようにしましょう。そして、`null` をエラー値に使うエラー処理は暗黙的なエラー状態をシステムのいたるところに持ち込むことになり、発見困難なバグを生む要因になります。後述しますが、Scala では Option というデータ構造を使うことでこの問題を解決します。

例外を投げる場合の注意点

Java のエラー処理で中心的な役割を果たすのが例外です。例外は今実行している処理を中断し、大域的に実行を移動できる便利な機能ですが、濫用することで処理の流れがわかりづらいコードにもなります。例外はエラー状態にのみ利用し、メソッドが正常な値を返す場合には使わないようにしましょう。

チェック例外の注意点

Java にはメソッドに `throws` 節を付けることで、メソッドを使う側に例外を処理することを強制するチェック例外という機能もあります。チェック例外は例外の発生を表現し、コンパイラにチェックさせるという点で便利な機能ですが、上げられた例外の catch 処理はわずらわしいものにもなりえます。使う側が適切に処理できない例外を上げられた場合はあまり意味のないエラー処理コードを書かざるをえません。よってチェック例外は利用側が catch して適切にエラー状態から回復できる場合にのみ利用したほうがいいでしょう。

例外翻訳の注意点

Java の例外は実装の変更により変化する場合があります。たとえば今まで HTTP で取得していたデータを MySQL に保存したとしましょう。その場合、今までは HTTPException が投げられていたものが、SQLException が投げられるようになるかもしれません。すると、この例外を catch する側も HTTPException ではなく SQLException を扱うようにしなければなりません。このように低レベルの実装の変更がプログラム全体に影響することがありえます。

そのような問題を防ぐために途中の層で一度例外を catch し、適切な例外で包んでもう一度投げる手法があります。このことを例外翻訳と呼びます。例外翻訳は例外に対する情報を増やし、catch する側の影響も少なくする手法です。ただし、この例外翻訳も乱用すると例外の種類が増えて例外処理が煩雑になる可能性もあるので注意が必要です。

例外をドキュメントに書く

例外はチェック例外でない場合、API から読み取ることができません。さらに後述しますが Scala ではチェック例外がないので、メソッドの型からどんな例外を投げるかは判別できません。そのため API ドキュメントには発生しうる例外についても書いておいたほうが良いでしょう。

例外の問題点

Java のエラー処理では例外が中心的な役割を担っていましたが、Scala でも例外は多く使われます。しかし、例外は便利な反面、様々な問題もあります。ここで例外の問題点を把握し、適切に使えるようになりましょう。

例外を使うとコントロールフローがわかりづらくなる

先ほど述べたように例外は、適切に使えば正常系の処理とエラー処理を分離し、コードの可読性を上げ、エラー処理をまとめる効果があります。しかし、往々にして例外の catch 漏れが発生し、障害に繋がることがあります。逆に例外を catch しているところで、どこで発生した例外を catch しているのか判別できないために、コードの修正を阻害する場合もあります。

例外は非同期プログラミングでは使えない

例外の動作は送出されたら catch されるまでコールスタックを遡っていくというものです。ということは別スレッドや、別のイベントループなどで実行される非同期プログラミングとは相容れないものです。特に Scala では非同期プログラミングが多用されるので、例外をそのまま使えないことが多いです。

例外は型チェックできない

チェック例外を使わない限り、どんな例外が発生するのかメソッドの型としては表現されません。また catch する側でも間違った例外をキャッチしているかどうかは実行時にしかわかりません。例外に頼りすぎると静的型付き言語の利点が損われます。

チェック例外の問題点

チェック例外を使わないとコンパイル時に型チェックできないわけですが、Scala では Java とは違いチェック例外の機能はなくなりました。これにはチェック例外の様々な問題点が理由としてあると思います

- 高階関数でチェック例外を扱うことが難しい
- ボイラープレートが増える
- 例外によるメソッド型の変更を防ぐために例外翻訳を多用せざるをえない

特に Scala では 1 番目の問題が大きいと思います。後述しますが、Scala ではチェック例外の代替手段として、エラーを表現するデータ型を使い、エラー処理を型安全にすることもできます。それらを考えると Scala でチェック例外をなくしたのは妥当な判断と言えるでしょう。

エラーを表現するデータ型を使った処理

例外に問題があるとすれば、どのようにエラーを扱えばよいのでしょうか。その答えの 1 つはエラーを例外ではなく、メソッドの返り値で返せるような値にすることです。

ここでは正常の値とエラー値のどちらかを表現できるデータ構造の紹介を通じて、Scala の関数型のエラー処理の方法を見ていきます。

Option

Option は Scala でもっとも多用されるデータ型の 1 つです。前述のとおり Java の null の代替として使われることが多いデータ型です。

Option 型は簡単に言うと、値を 1 つだけいれることのできるコンテナです。ただし、Option のまま様々なデータ変換処理ができるように便利な機能を持ちあわせています。

Option の作り方と値の取得

では具体的に Option の作り方と値の取得を見てみましょう。Option 型には具体的には

- `Some`
- `None`

以上 2 つの具体的な値が存在します。Some は何かしらの値が格納されている時の Option の型、None は値が何も格納されていない時の Option の型です。

具体的な動きを見てみましょう。Option に具体的な値が入った場合は以下の様な動きをします。

```
scala> val o: Option[String] = Option("hoge")
o: Option[String] = Some(hoge)

scala> o.get
res0: String = hoge

scala> o.isEmpty
res1: Boolean = false

scala> o.isDefined
res2: Boolean = true
```

今度は null を Option に入れるとどうなるでしょうか。

```
scala> val o: Option[String] = Option(null)
o: Option[String] = None

scala> o.isEmpty
res3: Boolean = true

scala> o.isDefined
res4: Boolean = false
```

```
scala> o.get
java.util.NoSuchElementException: None.get
  at scala.None$.get(Option.scala:347)
  at scala.None$.get(Option.scala:345)
  ... 946 elided
```

Option のコンパニオンオブジェクトの apply には引数が null であるかどうかのチェックが入っており、引数が null の場合、値が None になります。get メソッドを叩いた時に、java.util.NoSuchElementException という例外が起こっているので、これが NPE と同じだと思うかもしれません。しかし Option には以下の様な便利メソッドがあり、それらを回避することができます。

```
scala> o.getOrElse("")
res6: String = ""
```

以上は Option[String] の中身が None だった場合に、空文字を返すというコードになります。値以外にも処理を書くこともできます。

```
scala> o.getOrElse(throw new RuntimeException("null は受け入れられません"))
java.lang.RuntimeException: null は受け入れられません
```

```
  at $anonfun$1.apply(<console>:14)
  at $anonfun$1.apply(<console>:14)
  at scala.Option.getOrElse(Option.scala:121)
  ... 1014 elided
```

このように書くこともできるのです。

Option のパターンマッチ

上記では、手続き的に Option を処理しましたが、型を持っているためパターンマッチを使って処理することもできます。

```
scala> val s: Option[String] = Some("hoge")
s: Option[String] = Some(hoge)

scala> s match {
     |   case Some(str) => println(str)
     |   case None => throw new RuntimeException
     | }
hoge
```

上記のように Some か None にパターンマッチを行い、Some にパターンマッチする場合には、その中身の値を `str` という別の変数に束縛をすることも可能です。

中身を取りだすのではなく、中身を束縛するというテクニックは、`List` のパターンマッチでも行うことができますが、全く同様のことが Option でもできます。

Option に関数を適用する

Option には、コレクションの性質があると言いましたが、関数を内容の要素に適用できるという性質もそのまま持ち合わせています。

```
scala> Some(3).map(_ * 3)
res9: Option[Int] = Some(9)
```

このように、`map` で関数を適用する事もできます。なお、値が `None` の場合にはどうなるでしょうか。

```
scala> val n: Option[Int] = None
n: Option[Int] = None

scala> n.map(_ * 3)
res10: Option[Int] = None
```

None のままだと型情報を持たないので一度、変数にしていますが、None に 3 をかけるという関数を適用しても None のままです。この性質はとても便利で、その値が Option の中身が Some なのか None なのかどちらであったとしても、同様の処理で記述でき、処理を分岐させる必要がないのです。

Java 風に書くならば、

```
scala> if (n.isDefined) {
     |   n.get * 3
     | } else {
     |   throw new RuntimeException
     | }
java.lang.RuntimeException
  ... 1024 elided
```

きっと上記のように書くことになっていたでしょう。ただ、よくよく考えると上記の Java 風に書いた例と map の例は異なることに気が付きます。map では、値が Some の場合は中身に関数を適用しますが、None の時には何も実行しません。上記の例では例外を投げています。そして、値も Int 型の値を返していることも異なっています。

このように、None の場合に実行し、値を返す関数を定義できるのが `fold` です。`fold` の宣言を Scala の API ドキュメントから引用すると、

```
fold[B](ifEmpty: ⇒ B)(f: (A) ⇒ B): B
```

となります。

そして関数を適用した値を最終的に取得できます。

```
scala> n.fold(throw new RuntimeException)(_ * 3)
java.lang.RuntimeException
  at $anonfun$2.apply(<console>:14)
  at $anonfun$2.apply(<console>:14)
  at scala.Option.fold(Option.scala:158)
  ... 1021 elided
```

上記のように書くことで、None の際に実行する処理を定義し、かつ、関数を適用した中身の値を取得することができます。

```
scala> Some(3).fold(throw new RuntimeException)(_ * 3)
res13: Int = 9
```

`Some(3)` を与えるとこのように Int の 9 の値を返すことがわかります。

Option の入れ子を解消する

実際の複雑なアプリケーションの中では、Option の値が取得されることがよくあります。

たとえばキャッシュから情報を取得する場合は、キャッシュヒットする場合と、キャッシュミスする場合があり、それらは Scala ではよく Option 型で表現されます。

このようなキャッシュ取得が連続して繰り返された場合はどうなるでしょうか。例えば、1 つ目と 2 つ目の整数の値が Option で返ってきてそれをかけた値をもとめるような場合です。

```
scala> val v1: Option[Int] = Some(3)
v1: Option[Int] = Some(3)

scala> val v2: Option[Int] = Some(5)
v2: Option[Int] = Some(5)

scala> v1.map(i1 => v2.map(i2 => i1 * i2))
res14: Option[Option[Int]] = Some(Some(15))
```

map だけを使ってシンプルに実装するとこんな風になってしまいます。ウウッ…、悲しいことに `Option[Option[Int]]` のように Option が入れ子になってしまいます。

このような入れ子の option を解消するために用意されているのが、`flatten` です。

```
scala> v1.map(i1 => v2.map(i2 => i1 * i2)).flatten
res15: Option[Int] = Some(15)
```

最後に `flatten` を実行することで、Option の入れをを解消することができます。なお、これはちゃんと v2 が None である場合にも `flatten` は成立します。

```
scala> val v1: Option[Int] = Some(3)
v1: Option[Int] = Some(3)

scala> val v2: Option[Int] = None
v2: Option[Int] = None

scala> v1.map(i1 => v2.map(i2 => i1 * i2)).flatten
res16: Option[Int] = None
```

つまり、キャッシュミスで Some の値が取れなかった際も問題なくこの処理で動きます。

練習問題

`map` と `flatten` を利用して、`Some(2)` と `Some(3)` と `Some(5)` と `Some(7)` と `Some(11)` の値をかけて、`Some(2310)` を求めてみましょう。

```
val v1: Option[Int] = Some(2)
val v2: Option[Int] = Some(3)
val v3: Option[Int] = Some(5)
val v4: Option[Int] = Some(7)
val v5: Option[Int] = Some(11)
v1.map { i1 =>
    v2.map { i2 =>
        v3.map { i3 =>
            v4.map { i4 =>
                v5.map { i5 => i1 * i2 * i3 * i4 * i5 }
            }.flatten
        }.flatten
    }.flatten
}.flatten
```

flatMap

ここまでで、map と flatten を話しましたが、実際のプログラミングではこの両方を組み合わせて使うということが多々あります。そのためその 2 つを適用してくれる flatMap というメソッドが Option には用意されています。名前は flatMap なのですが、意味としては Option に map をかけて flatten を適用してくれます。

実際に先ほどの、Some(3) と Some(5) をかける例で利用してみると以下のようになります。

```
scala> val v1: Option[Int] = Some(3)
v1: Option[Int] = Some(3)

scala> val v2: Option[Int] = Some(5)
v2: Option[Int] = Some(5)

scala> v1.flatMap(i1 => v2.map(i2 => i1 * i2))
res18: Option[Int] = Some(15)
```

ずいぶんシンプルに書くことができるようになります。

Some(3) と Some(5) と Some(7) をかける場合はどうなるでしょうか。

```
scala> val v1: Option[Int] = Some(3)
v1: Option[Int] = Some(3)

scala> val v2: Option[Int] = Some(5)
v2: Option[Int] = Some(5)

scala> val v3: Option[Int] = Some(7)
v3: Option[Int] = Some(7)

scala> v1.flatMap(i1 => v2.flatMap(i2 => v3.map(i3 => i1 * i2 * i3)))
res19: Option[Int] = Some(105)
```

無論これは、v1, v2, v3 のいずれが None であった場合にも成立します。その場合には flatten の時と同様に None が最終的な答えになります。

```
scala> val v3: Option[Int] = None
v3: Option[Int] = None

scala> v1.flatMap(i1 => v2.flatMap(i2 => v3.map(i3 => i1 * i2 * i3)))
res20: Option[Int] = None
```

以上のようになります。

練習問題

flatMap を利用して、Some(2) と Some(3) と Some(5) と Some(7) と Some(11) の値をかけて、Some(2310) を求めてみましょう。

```
val v1: Option[Int] = Some(2)
val v2: Option[Int] = Some(3)
val v3: Option[Int] = Some(5)
val v4: Option[Int] = Some(7)
val v5: Option[Int] = Some(11)
v1.flatMap { i1 =>
    v2.flatMap { i2 =>
        v3.flatMap { i3 =>
            v4.flatMap { i4 =>
                v5.map { i5 => i1 * i2 * i3 * i4 * i5 }
            }
        }
    }
}
```

for を利用した flatMap のリファクタリング

Option はコレクションのようなものだという風に言いましたが、for を Option に使うこともできます。for 式は実際には `flatMap` と `map` 展開されて実行されるのです。

何をいっているのかわかりにくいと思いますので、先ほどの `Some(3)` と `Some(5)` と `Some(7)` を flatMap でかけるという処理を for で書いてみましょう。

```
scala> val v1: Option[Int] = Some(3)
v1: Option[Int] = Some(3)

scala> val v2: Option[Int] = Some(5)
v2: Option[Int] = Some(5)

scala> val v3: Option[Int] = Some(7)
v3: Option[Int] = Some(7)

scala> for { i1 <- v1
     |       i2 <- v2
     |       i3 <- v3 } yield i1 * i2 * i3
res22: Option[Int] = Some(105)
```

実はこの for 式は先ほどの `flatMap` と `map` で書かれたものとまったく同じ動作をします。`flatMap` と `map` を複数回使うような場合は for 式のほうがよりシンプルに書くことができていることがわかると思います。

練習問題

`for` を利用して、`Some(2)` と `Some(3)` と `Some(5)` と `Some(7)` と `Some(11)` の値をかけて、`Some(2310)` を求めてみましょう。

```
val v1: Option[Int] = Some(2)
val v2: Option[Int] = Some(3)
val v3: Option[Int] = Some(5)
val v4: Option[Int] = Some(7)
val v5: Option[Int] = Some(11)
for { i1 <- v1
      i2 <- v2
      i3 <- v3
      i4 <- v4
      i5 <- v5 } yield i1 * i2 * i3 * i4 * i5
```

Either

Option により null を使う必要はなくなりましたが、いっぽうで Option では処理が成功したかどうかしかわからないという問題があります。None の場合、値が取得できなかったことはわかりますが、エラーの状態は取得できないので、使用できるのはエラーの種類が問題にならないような場合のみです。

そんな Option と違い、エラー時にエラーの種類まで取得できるのが Either です。Option が正常な値と何もない値のどちらかを表現するデータ型だったのに対して、Either は 2 つの値のどちらかを表現するデータ型です。具体的には、Option では `Some` と `None` の 2 つの値を持ちましたが、Either は `Right` と `Left` の 2 つの値を持ちます。

```
scala> val v1: Either[String, Int] = Right(123)
v1: Either[String,Int] = Right(123)

scala> val v2: Either[String, Int] = Left("abc")
v2: Either[String,Int] = Left(abc)
```

パターンマッチで値を取得できるのも Option と同じです。

```
scala> v1 match {
     |   case Right(i) => println(i)
     |   case Left(s)  => println(s)
     | }
123
```

Either でエラー値を表現する

一般的に Either を使う場合、Left 値をエラー値、Right 値を正常な値とみなすことが多いです。英語の “right” が正しいという意味なので、それにかけているという説があります。そして Left に用いるエラー値ですが、これは代数的データ型（sealed trait と case class で構成される一連のデータと型のこと）で定義するとよいでしょう。パターンマッチの節で解説したように代数的データ型を用いることでエラーの処理が漏れているかどうかをコンパイラが検知してくれるようになります。単に `Throwable` 型をエラー型に使うのなら後述の Try で十分です。

例として Either を使ってログインのエラーを表現してみましょう。Left の値となる LoginError を定義します。sealed を使って代数的データ型として定義するのがポイントです。

```
sealed trait LoginError
// パスワードが間違っている場合のエラー
case object InvalidPassword extends LoginError
// name で指定されたユーザーが見つからない場合のエラー
case object UserNotFound extends LoginError
// パスワードがロックされている場合のエラー
case object PasswordLocked extends LoginError
```

ログイン API の型は以下のようにします。

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

object LoginService {
  def login(name: String, password: String): Either[LoginError, User] = ???
}
```

login メソッドはユーザー名とパスワードをチェックして正しい組み合わせの場合は User オブジェクトを Either の Right の値で返し、エラーが起きた場合は LoginError を Either の Left の値で返します。

それでは、この login メソッドを使ってみましょう。

```
LoginService.login(name = "dwango", password = "password") match {
  case Right(user) => println(s"id: ${user.id}")
  case Left(InvalidPassword) => println(s"Invalid Password!")
}
```

とりあえず呼び出して、println を使って中身を表示しているだけです。ここで注目していただきたいのが、Left の値のパターンマッチです。InvalidPassword の処理はしていますが、UserNotFound の場合と PasswordLocked の場合の処理が抜けてしまっています。そのような場合でもエラー値に代数的データ型を用いているので、コンパイラがエラー処理漏れを検知してくれます。

試しに上のコードをコンパイルしてみると、

```
<console>:11: warning: match may not be exhaustive.
It would fail on the following inputs: Left(PasswordLocked), Left(UserNotFound)
              LoginService.login(name = "dwango", password = "password") match {
                                 ^
```

のようにコンパイラが Left(PasswordLocked) と Left(UserNotFound) の処理が漏れていることを warning で教えてくれます。Either を使う場合はこのテクニックを覚えておいたほうがいいでしょう。

Either の map と flatMap

以上、見てきたように格納できるデータが増えているという点で Either は Option の拡張版に近いのですが、Scala の Either は map と flatMap の動作にちょっと癖があります。Scala の Either は Option とは違い、直接 map や flatMap メソッドを持ちません。Either オブジェクトのメソッドを見てみますと、

```
scala> val v: Either[String, Int] = Right(123)
v: Either[String,Int] = Right(123)

scala> v.
asInstanceOf   fold   isInstanceOf   isLeft   isRight   joinLeft   joinRight   left   right   swap   toString
```

`fold` や `isLeft` や `isRight` などは Option で同じようなメソッドがありましたが、肝心の map や flatMap がありません。これでは for 式を使って複数の Either を組み合わせることができません。

これには Scala の Either が左右の型を平等で扱っているという理由があります。たとえば自作の map メソッドを作ることを考えてみましょう。map メソッドは関数をコンテナの中身に適用できるようにするというものでした。Either の左右が平等に扱われると考えた場合、map メソッドは左右のどちらの値に適用すればいいのでしょうか？ この答えには 2 つのアプローチが考えられます。

- 暗黙的に左右どちらかのうち片方を優先し、そちらに関数を適用する（たとえば Right が正常な値になることが多いなら Right を暗黙的に優先するとか）
- 明示的に左右どちらを優先するか指定する

前者は Haskell のアプローチで、後者が Scala のアプローチになります。上記の Either のメソッドに `left` と `right` というものがありました。これが左右どちらを優先するのか決めるメソッドです。

試しに `right` メソッドを使ってみましょう。

```
scala> val v: Either[String, Int] = Right(123)
v: Either[String,Int] = Right(123)

scala> val e = v.right
e: scala.util.Either.RightProjection[String,Int] = RightProjection(Right(123))

scala> e.
asInstanceOf   canEqual   copy   e   exists   filter   flatMap   forall   foreach   get   getOrElse   isInstanceOf
```

Either に `right` メソッドを使ったら `RightProjection` というオブジェクトになりました。そして、この `RightProjection` オブジェクトを見てみると、ようやく List や Option で見慣れたようなメソッドが出てきました。これらのメソッドは Either の Right の値に対しておこなわれるメソッドということです。

ためしに `RightProjection` の `map` メソッドを使ってみましょう[*1]。

[*1] Scala2.11 以前だと `Product with Serializable with scala.util.Either` というような変な型になりますが、実

```
scala> val v: Either[String, Int] = Right(123)
v: Either[String,Int] = Right(123)

scala> v.right.map(_ * 2)
res25: Product with Serializable with scala.util.Either[String,Int] = Right(246)
```

これで map を使って値を二倍にする関数を Right に適用できました。ちなみに Either が Left の場合は何の処理もおこなわれません。これは Option で None に対して map を使った場合に何の処理もおこなわれないという動作に似ていますね。

注意してほしいのは RightProjection の map メソッドの返り値は Either であるという点です。つまり map や flatMap を連鎖して使う場合には毎回 Either を RightProjection に変化させる必要があるということです。

Option のときの掛け算の例を Either で書いてみると、

```
scala> val v1: Either[String, Int] = Right(3)
v1: Either[String,Int] = Right(3)

scala> val v2: Either[String, Int] = Right(5)
v2: Either[String,Int] = Right(5)

scala> val v3: Either[String, Int] = Right(7)
v3: Either[String,Int] = Right(7)

scala> for {
     |   i1 <- v1.right
     |   i2 <- v2.right
     |   i3 <- v3.right
     | } yield i1 * i2 * i3
res26: scala.util.Either[String,Int] = Right(105)
```

Right が正常な値として用いられることが多いとすれば、ほとんどの Either は RightProjection に変化させて使うことになります。この Scala の Either の仕様は Haskell の Either のような暗黙的に Right を優先するのに比べてわずらわしいと言われることもあります。とりあえず Either の map や flatMap などを使う場合は right で RightProjection に変化させると覚えてしまってよいと思います。

Try

Scala の Try は Either と同じように正常な値とエラー値のどちらかを表現するデータ型です。Either との違いは、2 つの型が平等ではなく、エラー値が Throwable に限定されており、型引数を 1 つしか取らないことです。具体的には Try は以下の 2 つの値をとります。

- Success

用上問題ないので、ひとまず気にしなくてよいです。この変な型になる問題は Scala2.12 以降では修正されています。https://github.com/scala/scala/pull/4355 https://issues.scala-lang.org/browse/SI-9173

- Failure

ここで Success は型変数を取り、任意の値を入れることができますが、Failure は Throwable しか入れることができません。そして Try には、コンパニオンオブジェクトの apply で生成する際に、例外を catch し、Failure にする機能があります。

```
scala> import scala.util.Try
import scala.util.Try

scala> val v: Try[Int] = Try(throw new RuntimeException("to be caught"))
v: scala.util.Try[Int] = Failure(java.lang.RuntimeException: to be caught)
```

この機能を使って、例外が起こりそうな箇所を Try で包み、Failure にして値として扱えるようにするのが Try の特徴です。

また Try は Either と違い、正常な値を片方に決めているので map や flatMap をそのまま使うことができます。

```
scala> val v1 = Try(3)
v1: scala.util.Try[Int] = Success(3)

scala> val v2 = Try(5)
v2: scala.util.Try[Int] = Success(5)

scala> val v3 = Try(7)
v3: scala.util.Try[Int] = Success(7)

scala> for {
     |   i1 <- v1
     |   i2 <- v2
     |   i3 <- v3
     | } yield i1 * i2 * i3
res27: scala.util.Try[Int] = Success(105)
```

NonFatal の例外

Try.apply が catch するのはすべての例外ではありません。NonFatal という種類の例外だけです。NonFatal ではないエラーはアプリケーション中で復旧が困難な非常に重度なものです。なので、NonFatal ではないエラーは catch せずにアプリケーションを終了させて、外部から再起動などをしたほうがいいです。

Try 以外でも、たとえば扱うことができる全ての例外をまとめて処理したい場合などに、

```
import scala.util.control.NonFatal

try {
  ???
} catch {
  case NonFatal(e) => // 例外の処理
```

```
}
```

というパターンが実践的なコード中に出てくることがしばしばあるので覚えておくとよいと思います。

Option と Either と Try の使い分け

ではエラー処理において Option と Either と Try はどのように使い分けるべきなのでしょうか。

まず基本的に Java で null を使うような場面は Option を使うのがよいでしょう。コレクションの中に存在しなかったり、ストレージ中から条件に合うものを発見できなかったりした場合は Option で十分だと考えられます。

次に Either ですが、Option を使うのでは情報が不足しており、かつ、エラー状態が代数的データ型としてちゃんと定められるものに使うのがよいでしょう。Java でチェック例外を使っていたようなところで使う、つまり、復帰可能なエラーだけに使うという考え方でもよいです。Either と例外を併用するのもアリだと思います。

Try は Java の例外をどうしても値として扱いたい場合に用いるとよいです。非同期プログラミングで使ったり、実行結果を保存しておき、あとで中身を参照したい場合などに使うことも考えられます。

Option の例外処理を Either でリファクタする実例

Scala でリレーショナルデータベースを扱う場合、関連をたどっていく中でどのタイミングで情報が取得できなかったのかを返さねばならないことがあります。

None を盲目的に処理するのであれば、flatMap や for 式をつかえば畳み込んでスッキリかけるのですが、関連を取得していくなかでどのタイミングで None が取得されてしまったのか返したい場合にはそうは行かず、結局 match case の深いネストになってしまいます。

例を挙げます。

ユーザーとアドレスがそれぞれデータベースに格納されており、ユーザー ID を利用してそのユーザーを検索し、ユーザーが持つアドレス ID でアドレスを検索し、さらにその郵便番号を取得するような場合を考えます。

失敗結果としては

- ユーザーがみつからない
- ユーザーがアドレスを持っていない
- アドレスがみつからない
- アドレスが郵便番号を持っていない

という 4 つの失敗パターンがあり、それらを結果オブジェクトとして返さなくてはなりません。

以下のようなコードになります。

```scala
object MainBefore {

  case class Address(id: Int, name: String, postalCode: Option[String])
  case class User(id: Int, name: String, addressId: Option[Int])

  val userDatabase: Map[Int, User] = Map (
    1 -> User(1, "太郎", Some(1)),
    2 -> User(2, "二郎", Some(2)),
    3 -> User(3, "プー太郎", None)
  )

  val addressDatabase: Map[Int, Address] = Map (
    1 -> Address(1, "渋谷", Some("150-0002")),
    2 -> Address(2, "国際宇宙ステーション", None)
  )

  sealed abstract class PostalCodeResult
  case class Success(postalCode: String) extends PostalCodeResult
  abstract class Failure extends PostalCodeResult
  case class UserNotFound() extends Failure
  case class UserNotHasAddress() extends Failure
  case class AddressNotFound() extends Failure
  case class AddressNotHasPostalCode() extends Failure

  // どこで None が生じたか取得しようとすると for 式がつかえず地獄のようなネストになる
  def getPostalCodeResult(userId: Int): PostalCodeResult = {
    findUser(userId) match {
      case Some(user) =>
        user.addressId match {
          case Some(addressId) =>
            findAddress(addressId) match {
              case Some(address) =>
                address.postalCode match {
                  case Some(postalCode) => Success(postalCode)
                  case None => AddressNotHasPostalCode()
                }
              case None => AddressNotFound()
            }
          case None => UserNotHasAddress()
        }
      case None => UserNotFound()
    }
  }

  def findUser(userId: Int): Option[User] = {
    userDatabase.get(userId)
  }

  def findAddress(addressId: Int): Option[Address] = {
    addressDatabase.get(addressId)
  }
```

```
  def main(args: Array[String]): Unit = {
    println(getPostalCodeResult(1)) // Success(150-0002)
    println(getPostalCodeResult(2)) // AddressNotHasPostalCode()
    println(getPostalCodeResult(3)) // UserNotHasAddress()
    println(getPostalCodeResult(4)) // UserNotFound()
  }
}
```

getPostalCodeResult が鬼のような match case のネストになっていることがわかります。このような可読性の低いコードを、Either を使って書きなおすことができます。

以下のように全ての find メソッドを Either で Failure を Left に、正常取得できた場合の値の型を Right にして書き直します。

find の各段階で Failure オブジェクトに引き換えるという動きをさせるわけです。

リファクタリングした結果は以下のようになります。

```
object MainRefactored {

  case class Address(id: Int, name: String, postalCode: Option[String])
  case class User(id: Int, name: String, addressId: Option[Int])

  val userDatabase: Map[Int, User] = Map (
    1 -> User(1, "太郎", Some(1)),
    2 -> User(2, "二郎", Some(2)),
    3 -> User(3, "プー太郎", None)
  )

  val addressDatabase: Map[Int, Address] = Map (
    1 -> Address(1, "渋谷", Some("150-0002")),
    2 -> Address(2, "国際宇宙ステーション", None)
  )

  sealed abstract class PostalCodeResult
  case class Success(postalCode: String) extends PostalCodeResult
  abstract class Failure extends PostalCodeResult
  case class UserNotFound() extends Failure
  case class UserNotHasAddress() extends Failure
  case class AddressNotFound() extends Failure
  case class AddressNotHasPostalCode() extends Failure

  // 本質的に何をしているかわかりやすくリファクタリング
  def getPostalCodeResult(userId: Int): PostalCodeResult = {
    (for {
      user <- findUser(userId).right
      address <- findAddress(user).right
      postalCode <- findPostalCode(address).right
    } yield Success(postalCode)).merge
  }

  def findUser(userId: Int): Either[Failure, User] = {
    userDatabase.get(userId).toRight(UserNotFound())
```

```
  }

  def findAddress(user: User): Either[Failure, Address] = {
    for {
      addressId <- user.addressId.toRight(UserNotHasAddress()).right
      address <- addressDatabase.get(addressId).toRight(AddressNotFound()).right
    } yield address
  }

  def findPostalCode(address: Address): Either[Failure, String] = {
    address.postalCode.toRight(AddressNotHasPostalCode())
  }

  def main(args: Array[String]): Unit = {
    println(getPostalCodeResult(1)) // Success(150-0002)
    println(getPostalCodeResult(2)) // AddressNotHasPostalCode()
    println(getPostalCodeResult(3)) // UserNotHasAddress()
    println(getPostalCodeResult(4)) // UserNotFound()
  }
}
```

以上のようになり、

```
  def getPostalCodeResult(userId: Int): PostalCodeResult = {
    (for {
      user <- findUser(userId).right
      address <- findAddress(user).right
      postalCode <- findPostalCode(address).right
    } yield Success(postalCode)).merge
  }
```

getPostalCodeResult が本質的に何をしているのかが非常にわかりやすいコードとなりました。何をしているかというと、Either では for 式を直接つかえないので.right というメソッドで、RightProjection という型にして、for 式が利用できる形に変換しています。そのあと、merge メソッドにより中身を畳み込んで取得しています。

第 15 章

implicit conversion（暗黙の型変換）と implicit parameter（暗黙のパラメータ）

Scala には、他の言語にはあまり見られない、implicit conversion と implicit parameter という機能があります。この 2 つの機能を上手く使いこなすことができれば、Scala でのプログラミングの生産性は劇的に向上するでしょう。なお、本当は、implicit conversion と implicit parameter は相互に関係がある 2 つの機能なのですが、今回学習する範囲では意識する必要がないと思われるので、2 つの独立した機能として学びます。

Implicit Conversion（★★★）

implicit conversion は暗黙の型変換機能をユーザが定義できるようにする機能です。

implicit conversion は

```
implicit def メソッド名(引数名: 引数の型): 返り値の型 = 本体
```

という形で定義します。implicit というキーワードがついていることと引数が 1 つしかない[*1]ことを除けば通常のメソッド定義と同様ですね。さて、implicit conversion では、引数の型と返り値の型に重要な意味があります。それは、これが、引数の型の式が現れたときに返り値の型を暗黙の変換候補として登録することになるからです。

定義した implicit conversion は大きく分けて二通りの使われ方をします。1 つは、新しく定義したユーザ定義の型などを既存の型に当てはめたい場合です。たとえば、

```
scala> implicit def intToBoolean(arg: Int): Boolean = arg != 0
warning: there was one feature warning; re-run with -feature for details
intToBoolean: (arg: Int)Boolean
```

[*1] 引数が 2 つ以上ある implicit def の定義も可能です。「implicit def のパラメーターに implicit が含まれる」という型クラス的な使い方をする場合は実際に implicit def に 2 つ以上のパラメーターが出現することがあります。ただしそういった定義は通常 implicit conversion とは呼ばれません

```
scala> if(1) {
     |   println("1 は真なり")
     | }
1 は真なり
```

といった形で、本来 Boolean しか渡せないはずの if 式に Int を渡すことができています。ただし、この使い方はあまり良いものではありません上の例をみればわかる通り、implicit conversion を定義することで、コンパイラにより、本来は if 式の条件式には Boolean 型の式しか渡せないようにチェックしているのを通りぬけることができてしまうからです。一部のライブラリではそのライブラリ内のデータ型と Scala 標準のデータ型を相互に変換するために、そのような implicit conversion を定義している例がありますが、本当にそのような変換が必要かよく考える必要があるでしょう。

pimp my library

もう 1 つの使い方は、pimp my library パターンと呼ばれ、既存のクラスにメソッドを追加して拡張する（ようにみせかける）使い方です。Scala 標準ライブラリの中にも大量の使用例があり、こちらが本来の使い方と言って良いでしょう。たとえば、これまでみたプログラムの中に (1 to 5) という式がありましたが、本来 Int 型は to というメソッドを持っていません。to メソッドは pimp my library パターンの使用例の最たるものです。コンパイラは、ある型に対するメソッド呼び出しを見つけたとき、そのメソッドを定義した型が implicit conversion の返り値の型にないか探索し、型が合ったら implicit conversion の呼び出しを挿入するのです。この使い方の場合、implicit conversion の返り値の型が他で使われるものでなければ安全に implicit conversion を利用することができます。

試しに、String の末尾に":-)"という文字列を追加して返す implicit conversion を定義してみましょう。

```
scala> class RichString(val src: String) {
     |   def smile: String = src + ":-)"
     | }
defined class RichString

scala> implicit def enrichString(arg: String): RichString = new RichString(arg)
warning: there was one feature warning; re-run with -feature for details
enrichString: (arg: String)RichString

scala> "Hi, ".smile
res1: String = Hi, :-)
```

ちゃんと文字列の末尾に":-)"を追加する smile メソッドが定義できています。さて、ここでひょっとしたら気がついた方もいるかもしれませんが、implicit conversion はそのままでは、既存のクラスへのメソッド追加のために使用するには冗長であるということです。Scala 2.10 からは、class に implicit というキーワードをつけることで同じようなことができるようになりました（皆さんが学習するのは Scala 2.11 なので気にする必要はありません。

Implicit Class

上の定義は、Scala 2.10 以降では、

```
scala> implicit class RichString(val src: String) {
     |   def smile: String = src + ":-)"
     | }
defined class RichString

scala> "Hi, ".smile
res0: String = Hi, :-)
```

という形で書きなおすことができます。 implicit class は pimp my library パターン専用の機能であり、implicit def で既存型への変換した場合などによる混乱がないため、Scala 2.10 以降で pimp my library パターンを使うときは基本的に後者の形式にすべきですが、サードパーティのライブラリや標準ライブラリでも前者の形式になっていることがあるので、そのようなコードも読めるようにしておきましょう。

練習問題

Int から Boolean への implicit conversion のように利用者を混乱させるようなものを考えて、定義してみてください。また、その implicit conversion にはどのような危険があるかを考えてください。

練習問題

pimp my library パターンで、既存のクラスの利用を便利にするような implicit conversion を 1 つ定義してみてください。それはどのような場面で役に立つでしょうか？

```
object Taps {
  implicit class Tap[T](self: T) {
    def tap[U](block: T => U): T = {
      block(self) //値は捨てる
      self
    }
  }

  def main(args: Array[String]): Unit = {
    "Hello, World".tap{s => println(s)}.reverse.tap{s => println(s)}
  }
}
```

```
scala> import Taps._
import Taps._

scala> Taps.main(Array())
Hello, World
dlroW ,olleH
```

メソッドチェインの中でデバッグプリントをはさみたいときに役に立ちます。

練習問題

Scala 標準ライブラリの中から pimp my library が使われている例を（先ほど挙げたものを除いて）1 つ以上見つけてください。

Implicit Parameter（★★★）

implicit parameter は主として 2 つの目的で使われます。1 つの目的は、あちこちのメソッドに共通で引き渡されるオブジェクト（たとえば、ソケットやデータベースのコネクションなど）を明示的に引き渡すのを省略するために使うものです。これは例で説明すると非常に簡単にわかると思います。

まず、データベースとのコネクションを表す Connection 型があるとします。データベースと接続するメソッドは全てこの Connection 型を引き渡さなければなりません。

```
def useDatabase1(...., conn: Connection)
def useDatabase2(...., conn: Connection)
def useDatabase3(...., conn: Connection)
```

この 3 つのメソッドは共通して Connection 型を引数に取るのに、呼びだす度に明示的に Connection オブジェクトを渡さなければならず面倒で仕方ありません。ここで implicit parameter の出番です。上のメソッド定義を

```
def useDatabase1(....)(implicit conn: Connection)
def useDatabase2(....)(implicit conn: Connection)
def useDatabase3(....)(implicit conn: Connection)
```

のように書き換えます。implicit 修飾子は引数の先頭の要素に付けなければならないという制約があり、implicit parameter を使うにはカリー化されたメソッド定義が必要になります。最後の引数リストが

```
(implicit conn: Connection)
```

とあるのがポイントです。Scala コンパイラは、このようにして定義されたメソッドが呼び出されると、現在のスコープからたどって直近の implicit とマークされた値を暗黙にメソッドに引き渡します。値を implicit としてマークするとは、たとえば次のようにして行います：

```
implicit val connection: Connection = connectDatabase(....)
```

このようにすることで、最後の引数リストに暗黙に Connection オブジェクトを渡してくれるのです。このような implicit parameter の使い方は Play 2 Framework や Scala の各種 O/R マッパーで頻出します。

implicit parameter のもう 1 つの使い方は、少々変わっています。まず、List の全ての要素の値を加算した結果を返す sum メソッドを定義したいとします。このメソッドはどのような定義になるでしょうか。ポイントは、「何の」List か全くわかっていないことで、整数の + メソッドをそのまま使ったりということはそのままではできないということです。このような場合、2 つの手順を踏みます。

まず、2 つの同じ型を足す（0 の場合はそれに相当する値を返す）方法を知っている型を定義します。ここではその型を Additive とします。Additive の定義は次のようになります：

```
trait Additive[A] {
  def plus(a: A, b: A): A
  def zero: A
}
```

ここで、Additive の型パラメータ A は加算される List の要素の型を表しています。また、

- zero: 型パラメータ A の **0** に相当する値を返す
- plus: 型パラメータ A を持つ 2 つの値を**加算して**返す

です。

次に、この Additive 型を使って、List の全ての要素を合計するメソッドを定義します：

```
def sum[A](lst: List[A])(m: Additive[A]) = lst.foldLeft(m.zero)((x, y) => m.plus(x, y))
```

後は、それぞれの型に応じた加算と 0 の定義を持った object を定義します。ここでは String と Int について定義をします。

```
object StringAdditive extends Additive[String] {
  def plus(a: String, b: String): String = a + b
  def zero: String = ""
}

object IntAdditive extends Additive[Int] {
  def plus(a: Int, b: Int): Int = a + b
  def zero: Int = 0
}
```

まとめると次のようになります。

```
trait Additive[A] {
  def plus(a: A, b: A): A
  def zero: A
}
```

```
object StringAdditive extends Additive[String] {
  def plus(a: String, b: String): String = a + b
  def zero: String = ""
}

object IntAdditive extends Additive[Int] {
  def plus(a: Int, b: Int): Int = a + b
  def zero: Int = 0
}

def sum[A](lst: List[A])(m: Additive[A]) = lst.foldLeft(m.zero)((x, y) => m.plus(x, y))
```

これで、Int 型の List も String 型の List のどちらの要素の合計も計算できる汎用的な sum メソッドができました。実際に呼び出したいときには、

```
scala> sum(List(1, 2, 3))(IntAdditive)
res6: Int = 6

scala> sum(List("A", "B", "C"))(StringAdditive)
res7: String = ABC
```

とすれば良いだけです。さて、これで目的は果たすことはできましたが、何の List の要素を合計するかは型チェックする時点ではわかっているのだからいちいち IntAdditive, StringAdditive を明示的に渡さずとも賢く推論してほしいものです。そして、まさにそれを implicit parameter で実現することができます。方法は簡単で、StringAdditive と IntAdditive の定義の前に implicit と付けることと、sum の最後の引数リストの m に implicit を付けるだけです。implicit parameter を使った最終形は次のようになります。

```
scala> trait Additive[A] {
     |   def plus(a: A, b: A): A
     |   def zero: A
     | }
defined trait Additive

scala> implicit object StringAdditive extends Additive[String] {
     |   def plus(a: String, b: String): String = a + b
     |   def zero: String = ""
     | }
defined object StringAdditive

scala> implicit object IntAdditive extends Additive[Int] {
     |   def plus(a: Int, b: Int): Int = a + b
     |   def zero: Int = 0
     | }
defined object IntAdditive

scala> def sum[A](lst: List[A])(implicit m: Additive[A]) = lst.foldLeft(m.zero)((x, y) => m.plus(x, y))
sum: [A](lst: List[A])(implicit m: Additive[A])A
```

```
scala> sum(List(1, 2, 3))
res8: Int = 6

scala> sum(List("A", "B", "C"))
res9: String = ABC
```

任意の List の要素の合計値を求める sum メソッドを自然な形（sum(List(1, 2, 3))）で呼びだすことができています。実は、implicit parameter のこのような使い方はプログラミング言語 Haskell から借りてきたもので、型クラス（を使った計算）と言われます。Haskell の用語だと、Additive を型クラス、StringAdditive と IntAdditive を Additive 型クラスのインスタンスと呼びます。

この implicit parameter の用法は標準ライブラリにもあって、たとえば、

```
scala> List[Int]().sum
res10: Int = 0

scala> List(1, 2, 3, 4).sum
res11: Int = 10

scala> List(1.1, 1.2, 1.3, 1.4).sum
res12: Double = 5.0
```

のように整数や浮動小数点数の合計値を特に気にとめることなく計算することができています。Scala において型クラスを定義・使用する方法を覚えると、設計の幅がグンと広がります。

練習問題

m: Additive[T] と値 t1: T, t2: T, t3: T は、次の条件を満たす必要があります。

```
m.plus(m.zero, t1) == t1  // 単位元
m.plus(t1, m.zero) == t1  // 単位元
m.plus(t1, m.plus(t2, t3)) == m.plus(m.plus(t1, t2), t3) // 結合則
```

このような条件を満たす型 T と単位元 zero、演算 plus を探し出し、Additive[T] を定義しましょう。この際、条件が満たされていることをいくつかの入力に対して確認してみましょう。また、定義した Additive[T] を implicit にして、T の合計値を先ほどの sum で計算できることを確かめてみましょう。

ヒント：このような条件を満たすものは無数にありますが、思いつかない人はたとえば x 座標と y 座標からなる点を表すクラス Point を考えてみると良いでしょう。

```
object Additives {
  trait Additive[A] {
    def plus(a: A, b: A): A
    def zero: A
  }
```

```
  implicit object StringAdditive extends Additive[String] {
    def plus(a: String, b: String): String = a + b
    def zero: String = ""
  }

  implicit object IntAdditive extends Additive[Int] {
    def plus(a: Int, b: Int): Int = a + b
    def zero: Int = 0
  }

  case class Point(x: Int, y: Int)

  implicit object PointAdditive extends Additive[Point] {
    def plus(a: Point, b: Point): Point = Point(a.x + b.x, a.y + b.y)
    def zero: Point = Point(0, 0)
  }

  def sum[A](lst: List[A])(implicit m: Additive[A]) = lst.foldLeft(m.zero)((x, y) => m.plus(x, y))
}
```

```
scala> import Additives._
import Additives._

scala> println(sum(List(Point(1, 1), Point(2, 2), Point(3, 3)))) // Point(6, 6)
Point(6,6)

scala> println(sum(List(Point(1, 2), Point(3, 4), Point(5, 6)))) // Point(9, 12)
Point(9,12)
```

練習問題

`List[Int]` と `List[Double]` の sum を行うために、標準ライブラリでは何という型クラス（1 つ）と型クラスのインスタンス（2 つ）を定義しているかを、Scala 標準ライブラリから探して挙げなさい。

型クラス：

- Numeric[T]

型クラスのインスタンス：

- IntIsIntegral
- DoubleAsIfIntegral

implicit の探索範囲

implicit def や implicit parameter の値が探索される範囲には、

- ローカルで定義されたもの
- import で指定されたもの
- スーパークラスで定義されたもの
- コンパニオンオブジェクトで定義されたもの

などがあります。この中で注目していただきたいのが、コンパニオンオブジェクトで implicit の値を定義するパターンです。

たとえば新しく Rational（有理数）型を定義したとして、コンパニオンオブジェクトに先ほど使った Additive 型クラスのインスタンスを定義しておきます。

```
case class Rational(num: Int, den: Int)

object Rational {
  implicit object RationalAdditive extends Additive[Rational] {
    def plus(a: Rational, b: Rational): Rational = {
      if (a == zero) {
        b
      } else if (b == zero) {
        a
      } else {
        Rational(a.num * b.den + b.num * a.den, a.den * b.den)
      }
    }
    def zero: Rational = Rational(0, 0)
  }
}
```

すると、import をしていないのに、この Additive 型クラスのインスタンスを使うことができます。

```
scala> sum(List(Rational(1, 1), Rational(2, 2)))
res0: Rational = Rational(4,2)
```

新しくデータ型を定義し、型クラスインスタンスも一緒に定義したい場合によく出てくるパターンなので覚えておくとよいでしょう。

第 16 章

型クラスの紹介

本章では Implicit の章で説明した型クラスの具体例を紹介します。本章で紹介する型クラスは、必ずしも Scala でのプログラミングに必要というわけではありません。しかし、世の中に存在する Scala で実装されたライブラリやアプリケーションのいくつかでは、本章で紹介する型クラスなどを多用している場合があります。そのようなライブラリやアプリケーションに出会った際にも臆さずコードリーディングができるよう、最低限の知識をつけることが本章の目的です。

本章で紹介する型クラスを絡めた Scala でのプログラミングについて詳しく知りたい場合は Scala 関数型デザイン＆プログラミングを読みましょう。

Functor（★★）

前章に登場した `List` や `Option` には、`map` という関数が共通して定義されていました。この `map` 関数がある規則を満たす場合は Functor 型クラスとして抽象化できます[*1]。

```
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
}
```

この型クラスが満たすべき規則は 2 つです。

```
def identityLaw[F[_], A](fa: F[A])(implicit F: Functor[F]): Boolean =
  F.map(fa)(identity) == fa

def compositeLaw[F[_], A, B, C](fa: F[A], f1: A => B, f2: B => C)(implicit F: Functor[F]): Boolean =
  F.map(fa)(f2 compose f1) == F.map(F.map(fa)(f1))(f2)
```

例として、Option 型で Functor 型クラスのインスタンスを定義し、前述の規則を満たすかどうか調べてみましょう。

[*1] ここで出現する `F` は、通常の型ではなく、「何かの型を受け取って、型を返すもの」で、型構築子、型コンストラクタなどと呼びます。`List` や `Option` は型構築子の一種です。詳細については、型システム入門プログラミング言語と型の理論の第 VI 部「高階の型システム」を参照してください。

```
scala> import scala.language.higherKinds
import scala.language.higherKinds

scala> trait Functor[F[_]] {
     |   def map[A, B](fa: F[A])(f: A => B): F[B]
     | }
defined trait Functor

scala> def identityLaw[F[_], A](fa: F[A])(implicit F: Functor[F]): Boolean =
     |   F.map(fa)(identity) == fa
identityLaw: [F[_], A](fa: F[A])(implicit F: Functor[F])Boolean

scala> def compositeLaw[F[_], A, B, C](fa: F[A], f1: A => B, f2: B => C)(implicit F: Functor[F]): Boolean =
     |   F.map(fa)(f2 compose f1) == F.map(F.map(fa)(f1))(f2)
compositeLaw: [F[_], A, B, C](fa: F[A], f1: A => B, f2: B => C)(implicit F: Functor[F])Boolean

scala> implicit object OptionFunctor extends Functor[Option] {
     |   def map[A, B](fa: Option[A])(f: A => B): Option[B] = fa.map(f)
     | }
defined object OptionFunctor

scala> val n: Option[Int] = Some(2)
n: Option[Int] = Some(2)

scala> identityLaw(n)
res1: Boolean = true

scala> compositeLaw(n, (i: Int) => i * i, (i: Int) => i.toString)
res2: Boolean = true
```

Applicative Functor（★★）

複数の値が登場する場合には Functor では力不足です。そこで、複数の引数を持つ関数と値を組み合わせて 1 つの値を作りだせる機能を提供する Applicative Functor が登場します。

```
trait Applicative[F[_]] {
  def point[A](a: A): F[A]
  def ap[A, B](fa: F[A])(f: F[A => B]): F[B]
}
```

Applicative Functor は Functor を特殊化したものなので、Applicative Functor が持つ関数から map 関数を定義できます。

```
def map[F[_], A, B](fa: F[A])(f: A => B)(implicit F: Applicative[F]): F[B] =
  F.ap(fa)(F.point(f))
```

Applicative Functor が満たすべき規則は以下の通りです。

```
def identityLaw[F[_], A](fa: F[A])(implicit F: Applicative[F]): Boolean =
  F.ap(fa)(F.point((a: A) => a)) == fa

def homomorphismLaw[F[_], A, B](f: A => B, a: A)(implicit F: Applicative[F]): Boolean =
  F.ap(F.point(a))(F.point(f)) == F.point(f(a))

def interchangeLaw[F[_], A, B](f: F[A => B], a: A)(implicit F: Applicative[F]): Boolean =
  F.ap(F.point(a))(f) == F.ap(f)(F.point((g: A => B) => g(a)))
```

また、ap と point を使って定義した map 関数が Functor のものと同じ振る舞いになることを確認する必要があります。

例として、Option 型で Applicative Functor を定義してみましょう。

```
scala> trait Applicative[F[_]] {
     |   def point[A](a: A): F[A]
     |   def ap[A, B](fa: F[A])(f: F[A => B]): F[B]
     |   def map[A, B](fa: F[A])(f: A => B): F[B] = ap(fa)(point(f))
     | }
defined trait Applicative

scala> def identityLaw[F[_], A](fa: F[A])(implicit F: Applicative[F]): Boolean =
     |   F.ap(fa)(F.point((a: A) => a)) == fa
identityLaw: [F[_], A](fa: F[A])(implicit F: Applicative[F])Boolean

scala> def homomorphismLaw[F[_], A, B](f: A => B, a: A)(implicit F: Applicative[F]): Boolean =
     |   F.ap(F.point(a))(F.point(f)) == F.point(f(a))
homomorphismLaw: [F[_], A, B](f: A => B, a: A)(implicit F: Applicative[F])Boolean

scala> def interchangeLaw[F[_], A, B](f: F[A => B], a: A)(implicit F: Applicative[F]): Boolean =
     |   F.ap(F.point(a))(f) == F.ap(f)(F.point((g: A => B) => g(a)))
interchangeLaw: [F[_], A, B](f: F[A => B], a: A)(implicit F: Applicative[F])Boolean

scala> implicit object OptionApplicative extends Applicative[Option] {
     |   def point[A](a: A): Option[A] = Some(a)
     |   def ap[A, B](fa: Option[A])(f: Option[A => B]): Option[B] = f match {
     |     case Some(g) => fa match {
     |       case Some(a) => Some(g(a))
     |       case None => None
     |     }
     |     case None => None
     |   }
     | }
defined object OptionApplicative

scala> val a: Option[Int] = Some(1)
a: Option[Int] = Some(1)

scala> val f: Int => String = { i => i.toString }
f: Int => String = <function1>

scala> val af: Option[Int => String] = Some(f)
```

```
af: Option[Int => String] = Some(<function1>)

scala> identityLaw(a)
res5: Boolean = true

scala> homomorphismLaw(f, 1)
res6: Boolean = true

scala> interchangeLaw(af, 1)
res7: Boolean = true

scala> OptionApplicative.map(a)(_ + 1) == OptionFunctor.map(a)(_ + 1)
res8: Boolean = true
```

Monad（★★）

ある値を受け取りその値を包んだ型を返す関数を Applicative Functor で扱おうとすると、型がネストしてしまい平坦化できません。このネストする問題を解決するために Monad と呼ばれる型クラスを用います。

```
trait Monad[F[_]] {
  def point[A](a: A): F[A]
  def bind[A, B](fa: F[A])(f: A => F[B]): F[B]
}
```

bind は Option や List で登場した flatMap を抽象化したものです。
Monad は以下の規則を満たす必要があります。

```
def rightIdentityLaw[F[_], A](a: F[A])(implicit F: Monad[F]): Boolean =
  F.bind(a)(F.point(_)) == a

def leftIdentityLaw[F[_], A, B](a: A, f: A => F[B])(implicit F: Monad[F]): Boolean =
  F.bind(F.point(a))(f) == f(a)

def associativeLaw[F[_], A, B, C](fa: F[A], f: A => F[B], g: B => F[C])(implicit F: Monad[F]): Boolean =
  F.bind(F.bind(fa)(f))(g) == F.bind(fa)((a: A) => F.bind(f(a))(g))
```

Monad は Applicative Functor を特殊化したものなので、Monad が持つ関数から point 関数と ap 関数を定義できます。point に関しては同じシグネチャなので自明でしょう。

```
def ap[F[_], A, B](fa: F[A])(f: F[A => B])(implicit F: Monad[F]): F[B] =
  F.bind(f)((g: A => B) => F.bind(fa)((a: A) => F.point(g(a))))
```

それでは、Option 型が前述の規則をみたすかどうか確認してみましょう。

```
scala> trait Monad[F[_]] {
     |   def point[A](a: A): F[A]
     |   def bind[A, B](fa: F[A])(f: A => F[B]): F[B]
```

```
     | }
defined trait Monad

scala> def rightIdentityLaw[F[_], A](a: F[A])(implicit F: Monad[F]): Boolean =
     |   F.bind(a)(F.point(_)) == a
rightIdentityLaw: [F[_], A](a: F[A])(implicit F: Monad[F])Boolean

scala> def leftIdentityLaw[F[_], A, B](a: A, f: A => F[B])(implicit F: Monad[F]): Boolean =
     |   F.bind(F.point(a))(f) == f(a)
leftIdentityLaw: [F[_], A, B](a: A, f: A => F[B])(implicit F: Monad[F])Boolean

scala> def associativeLaw[F[_], A, B, C](fa: F[A], f: A => F[B], g: B => F[C])(implicit F: Monad[F]): Boolean =
     |   F.bind(F.bind(fa)(f))(g) == F.bind(fa)((a: A) => F.bind(f(a))(g))
associativeLaw: [F[_], A, B, C](fa: F[A], f: A => F[B], g: B => F[C])(implicit F: Monad[F])Boolean

scala> implicit object OptionMonad extends Monad[Option] {
     |   def point[A](a: A): Option[A] = Some(a)
     |   def bind[A, B](fa: Option[A])(f: A => Option[B]): Option[B] = fa match {
     |     case Some(a) => f(a)
     |     case None => None
     |   }
     | }
defined object OptionMonad

scala> val fa: Option[Int] = Some(1)
fa: Option[Int] = Some(1)

scala> val f: Int => Option[Int] = { n => Some(n + 1) }
f: Int => Option[Int] = <function1>

scala> val g: Int => Option[Int] = { n => Some(n * n) }
g: Int => Option[Int] = <function1>

scala> rightIdentityLaw(fa)
res11: Boolean = true

scala> leftIdentityLaw(1, f)
res12: Boolean = true

scala> associativeLaw(fa, f, g)
res13: Boolean = true
```

Monoid（★★）

2 つの同じ型を結合する機能を持ち、更にゼロ値を知る型クラスは Monoid と呼ばれています。

```
trait Monoid[F] {
  def append(a: F, b: F): F
  def zero: F
}
```

前章で定義した Additive 型とよく似ていますが、Monoid は次の規則を満たす必要があります。

```
def leftIdentity[F](a: F)(implicit F: Monoid[F]): Boolean = a == F.append(F.zero, a)
def rightIdentity[F](a: F)(implicit F: Monoid[F]): Boolean = a == F.append(a, F.zero)
def associativeLaw[F](a: F, b: F, c: F)(implicit F: Monoid[F]): Boolean = {
  F.append(F.append(a, b), c) == F.append(a, F.append(b, c))
}
```

Option[Int] 型で Monoid インスタンスを定義してみましょう。

```
scala> trait Monoid[F] {
     |   def append(a: F, b: F): F
     |   def zero: F
     | }
defined trait Monoid

scala> def leftIdentity[F](a: F)(implicit F: Monoid[F]): Boolean = a == F.append(F.zero, a)
leftIdentity: [F](a: F)(implicit F: Monoid[F])Boolean

scala> def rightIdentity[F](a: F)(implicit F: Monoid[F]): Boolean = a == F.append(a, F.zero)
rightIdentity: [F](a: F)(implicit F: Monoid[F])Boolean

scala> def associativeLaw[F](a: F, b: F, c: F)(implicit F: Monoid[F]): Boolean = {
     |   F.append(F.append(a, b), c) == F.append(a, F.append(b, c))
     | }
associativeLaw: [F](a: F, b: F, c: F)(implicit F: Monoid[F])Boolean

scala> implicit object OptionIntMonoid extends Monoid[Option[Int]] {
     |   def append(a: Option[Int], b: Option[Int]): Option[Int] = (a, b) match {
     |     case (None, None) => None
     |     case (Some(v), None) => Some(v)
     |     case (None, Some(v)) => Some(v)
     |     case (Some(v1), Some(v2)) => Some(v1 + v2)
     |   }
     |   def zero: Option[Int] = None
     | }
defined object OptionIntMonoid

scala> val n: Option[Int] = Some(1)
n: Option[Int] = Some(1)

scala> val m: Option[Int] = Some(2)
m: Option[Int] = Some(2)

scala> val o: Option[Int] = Some(3)
o: Option[Int] = Some(3)

scala> leftIdentity(n)
res14: Boolean = true

scala> rightIdentity(n)
res15: Boolean = true

scala> associativeLaw(n, m, o)
```

```
res16: Boolean = true
```

型によっては結合方法が複数存在する場合があります。その際は複数の Monoid インスタンスを定義しておき、状況に応じて使いたい Monoid インスタンスを選択できるようにしておきましょう。

第 17 章

Future/Promise について（★★★）

Future と Promise は非同期プログラミングにおいて、終了しているかどうかわからない処理結果を抽象化した型です。Future は未来の結果を表す型です。Promise は一度だけ、成功あるいは失敗を表す、処理または値を設定することで Future に変換できる型です。

JVM 系の言語では、マルチスレッドで並行処理を使った非同期処理を行うことが多々あります。無論ブラウザ上の JavaScript のようなシングルスレッドで行うような非同期処理もありますが、マルチスレッドで行う非同期処理は定義した処理群が随時行われるのではなく、マルチコアのマシンならば大抵の場合、複数の CPU で別々に実行されることとなります。

具体的に非同期処理が行われている例としては、UI における読み込み中のインジゲーターなどがあげられます。読み込み中のインジゲーターがアニメーションしている間も、ダイアログを閉じたり、別な操作をすることができるのは、読み込み処理が非同期でおこなわれているからです。

なお、このような特定のマシンの限られたリソースの中で、マルチスレッドやマルチプロセスによって順不同もしくは同時に処理を行うことを、並行（Concurrent）処理といいます。マルチスレッドの場合はプロセスとメモリ空間とファイルハンドラを複数のスレッドで共有し、マルチプロセスの場合はメモリ管理は別ですが CPU リソースを複数のプロセスで共有しています。（注、スレッドおよびプロセスのような概念については知っているものとみなして説明していますのでご了承ください）

リソースが共有されているかどうかにかかわらず、完全に同時に処理を行っていくことを、並列（Parallel）処理といいます。大抵の場合、複数のマシンで分散実行させるような分散系を利用したスケールするような処理を並列処理系と呼びます。

このたびはこのような並行処理を使った非同期処理を行った場合に、とても便利な Future と Promise というそれぞれのクラスの機能と使い方について説明を行います。

Future とは（★★★）

Future とは、非同期に処理される結果が入った Option 型のようなものです。その処理が終わっているのかどうかや（isCompleted）、正しく終わった時の処理を適用する（onSuccess）、例外が起こったときの処理を適用する（onFailure）といったことが可能な他、flatMap や filter、for 式の適

用といったような Option や List でも利用できる性質も持ち合わせています。

ライブラリやフレームワークの処理が非同期主体となっている場合、この Future は基本的で重要な役割を果たすクラスとなります。

なお Java にも Future というクラスがありますが、こちらには関数を与えたり[*1]、Option の持つ特性はありません。また、ECMAScript 6 にある Promise という機能がありますが、そちらの方が Scala の Future の機能に似ています。この ECMAScript 6 の Promise と Scala の Promise は、全く異なる機能であるため注意が必要です。

実際のコード例を見てみましょう。

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

object FutureSample extends App {

  val s = "Hello"
  val f: Future[String] = Future {
    Thread.sleep(1000)
    s + " future!"
  }

  f.onSuccess { case s: String =>
    println(s)
  }

  println(f.isCompleted) // false

  Thread.sleep(5000) // Hello future!

  println(f.isCompleted) // true
}
```

出力結果は、

```
false
Hello future!
true
```

のようになります。

以上は Future 自体の機能を理解するためのサンプルコードです。非同期プログラミングは、sbt console で実装するのが難しいのでファイルに書かせてもらいました。Future シングルトンは関数を与えるとその関数を非同期に与える `Future[+T]` を返します。上記の実装例ではまず、1000 ミリ秒待機して、`"Hello"`と`" future!"`を文字列結合するという処理を非同期に処理します。そして成功時の処理を定義した後 future が処理が終わっているかを確認し、future の結果取得を 5000 ミリ秒間

[*1] ただし、Java 8 から追加された java.util.concurrent.Future のサブクラスである CompletableFuture には、関数を引数にとるメソッドがあります。

待つという処理を行った後、その結果がどうなっているのかをコンソールに出力するという処理をします。

なお以上のように 5000 ミリ秒待つという他に、その Future 自体の処理を待つという書き方もすることができます。`Thread.sleep(5000)` を `Await.ready(f, 5000 millisecond)` とすることで、Future が終わるまで最大 5000 ミリ秒を待つという書き方となります。ただし、この書き方をする前に、

```
import scala.concurrent.Await
import scala.concurrent.duration._
import scala.language.postfixOps
```

以上を import 文に追加する必要があります。さらにこれらがどのように動いているのかを、スレッドの観点から見てみましょう。以下のようにコードを書いてみます。

```
import scala.concurrent.{Await, Future}
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration._
import scala.language.postfixOps

object FutureSample extends App {

  val s = "Hello"
  val f: Future[String] = Future {
    Thread.sleep(1000)
    println(s"[ThreadName] In Future: ${Thread.currentThread.getName}")
    s + " future!"

  }

  f.onSuccess { case s: String =>
    println(s"[ThreadName] In onSuccess: ${Thread.currentThread.getName}")
    println(s)
  }

  println(f.isCompleted) // false

  Await.ready(f, 5000 millisecond) // Hello future!

  println(s"[ThreadName] In App: ${Thread.currentThread.getName}")
  println(f.isCompleted) // true
}
```

この実行結果については、

```
false
[ThreadName] In Future: ForkJoinPool-1-worker-5
[ThreadName] In App: main
true
```

```
[ThreadName] In onSuccess: ForkJoinPool-1-worker-5
Hello future!
```

となります。以上のコードではそれぞれのスレッド名を各箇所について出力してみました。非常に興味深い結果ですね。Future と onSuccess に渡した関数に関しては、ForkJoinPool-1-worker-5 という main スレッドとは異なるスレッドで実行されています。

つまり Future を用いることで知らず知らずのうちのマルチスレッドのプログラミングが実行されていたということになります。また、Await.ready(f, 5000 millisecond) で処理を書いたことで、isCompleted の確認処理のほうが、"Hello future!"の文字列結合よりも先に出力されていることがわかります。これは文字列結合の方が値参照よりもコストが高いためこのようになります。

ForkJoinPool に関しては、Java の並行プログラミングをサポートする ExecutorService というインタフェースを被ったクラスとなります。内部的にスレッドプールを持っており、スレッドを使いまわすことによって、スレッドを作成するコストを低減し高速化を図っています。

Future についての動きがわかった所で、Future が Option のように扱えることも説明します。

```
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future
import scala.util.{Failure, Random, Success}

object FutureOptionUsageSample extends App {
  val random = new Random()
  val waitMaxMilliSec = 3000

  val futureMilliSec: Future[Int] = Future {
    val waitMilliSec = random.nextInt(waitMaxMilliSec);
    if(waitMilliSec < 1000) throw new RuntimeException(s"waitMilliSec is ${waitMilliSec}" )
    Thread.sleep(waitMilliSec)
    waitMilliSec
  }

  val futureSec: Future[Double] = futureMilliSec.map(i => i.toDouble / 1000)

  futureSec onComplete {
    case Success(waitSec) => println(s"Success! ${waitSec} sec")
    case Failure(t) => println(s"Failure: ${t.getMessage}")
  }

  Thread.sleep(3000)
 }
```

出力例としては、Success! 1.538 sec や Failure: waitMilliSec is 971 というものになります。この処理では、3000 ミリ秒を上限としたランダムな時間を待ってその待ったミリ秒を返す Future を定義しています。ただし、1000 ミリ秒未満しか待たない場合には失敗とみなし例外を投げます。この最初にえられる Future を futureMilliSec としていますが、その後、map メソッドを利用して Int のミリ秒を Doubule の秒に変換しています。なお先ほどと違ってこの度は、onSuccess ではなく onComplete を利用して成功と失敗の両方の処理を記述しました。

以上の実装のように Future は結果を Option のように扱うことができるわけです。無論 map も使えますが Option がネストしている場合に flatMap を利用できるのと同様に、flatMap も Future に対して利用することもできます。つまり map の中での実行関数がさらに Future を返すような場合も問題なく Future を利用していけるのです。`val futureSec: Future[Double] = futureMilliSec.map(i => i.toDouble / 1000)` を上記のミリ秒を秒に変換する部分を 100 ミリ秒はかかる非同期の Future にしてみた例は以下のとおりです。

```
val futureSec: Future[Double] = futureMilliSec.flatMap(i => Future {
  Thread.sleep(100)
  i.toDouble / 1000
})
```

map で適用する関数で Option がとれてきてしまうのを flatten できるという書き方と同じように、Future に適用する関数の中でさらに Future が取得できるような場合では、flatMap が適用できます。この書き方のお陰で非常に複雑な非同期処理を、比較的シンプルなコードで表現してやることができるようになります。

Future を使って非同期に取れてくる複数の結果を利用して結果を作る

さて、flatMap が利用できるということは、for 式も利用できます。これらはよく複数の Future を組み合わせて新しい Future を作成するのに用いられます。実際に実装例を見てみましょう。

```
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future
import scala.language.postfixOps
import scala.util.{Failure, Success, Random}

object CompositeFutureSample extends App {
  val random = new Random()
  val waitMaxMilliSec = 3000

  def waitRandom(futureName: String): Int = {
    val waitMilliSec = random.nextInt(waitMaxMilliSec);
    if(waitMilliSec < 500) throw new RuntimeException(s"${futureName} waitMilliSec is ${waitMilliSec}" )
    Thread.sleep(waitMilliSec)
    waitMilliSec
  }

  val futureFirst: Future[Int] = Future { waitRandom("first") }
  val futureSecond: Future[Int] = Future { waitRandom("second") }

  val compositeFuture: Future[(Int, Int)] = for {
    first: Int <- futureFirst
    second: Int <- futureSecond
  } yield (first, second)
```

```
  compositeFuture onComplete  {
    case Success((first, second)) => println(s"Success! first:${first} second:${second}")
    case Failure(t) => println(s"Failure: ${t.getMessage}")
  }

  Thread.sleep(5000)
 }
```

先ほど紹介した例に似ていますが、ランダムで生成した最大 3 秒間待つ関数を用意し、500 ミリ秒未満しか待たなかった場合は失敗とみなします。その関数を実行する関数を Future として 2 つ用意し、それらを for 式で畳み込んで新しい Future を作っています。そして最終的に新しい Future に対して成功した場合と失敗した場合を出力します。

出力結果としては、`Success! first:1782 second:1227` や `Failure: first waitMilliSec is 412` や `Failure: second waitMilliSec is 133` といったものとなります。

なお Future には filter の他、様々な並列実行に対するメソッドが存在しますので、API ドキュメントを見てみてください。また複数の Future 生成や並列実行に関してのまとめられた日本語の記事もありますので、複雑な操作を試してみたい際にはぜひ参考にしてみてください。

Promise とは（★）

Promise とは、

一度だけ、成功あるいは失敗を表す、処理または値を設定することによって、Future に変換することのできるクラスです。Promise は、一見可変オブジェクトのような振る舞いをします。なかなかわかりにくいかと思いますので、実際にサンプルコードを示します。

```
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.{Promise, Future}
import scala.util.{Success, Failure, Random}

object PromiseSample extends App {
  val random = new Random()
  val promiseGetInt: Promise[Int] = Promise[Int]

  val futureGetInt: Future[Int] = promiseGetInt.success(1).future

  futureGetInt.onComplete {
    case Success(i) => println(s"Success! i: ${i}")
    case Failure(t) => println(s"Failure! t: ${t.getMessage}")
  }

  Thread.sleep(1000)
}
```

この処理は必ず `Success! i: 1` という値を返します。`promiseGetInt.success(1).future` を `promiseGetInt.future` のように成功結果を与えないような処理にした場合には、onComplete が呼ばれることはないため、何も出力されません。

なおこの 1 度だけしか結果が適用されないという特性を活かして、Future を組み合わせてものが実装できます。複数 success が定義される場合には、success メソッドの場合に trySuccess という Promise のメソッドを利用します。success を利用して複数回成功した値を定義した場合には、例外 `IllegalStateException` が投げられます。では早速実装例を見てみましょう。

```
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.{Future, Promise}
import scala.util.{Failure, Random, Success}

object PromiseFutureCompositionSample extends App {
  val random = new Random()
  val promiseGetInt: Promise[Int] = Promise[Int]

  val firstFuture: Future[Int] = Future {
    Thread.sleep(100)
    1
  }
  firstFuture.onSuccess{ case i => promiseGetInt.trySuccess(i)}

  val secondFuture: Future[Int] = Future {
    Thread.sleep(200)
    2
  }
  secondFuture.onSuccess{ case i => promiseGetInt.trySuccess(i)}

  val futureGetInt: Future[Int] = promiseGetInt.future

  futureGetInt.onComplete {
    case Success(i) => println(s"Success! i: ${i}")
    case Failure(t) => println(s"Failure! t: ${t.getMessage}")
  }

  Thread.sleep(1000)
}
```

結果は必ず、`Success! i: 1` が表示されます。100 ミリ秒待って 1 を返す firstFuture と、200 ミリ秒待って 2 を返す `secondFuture` が定義されています。時系列的に、`firstFuture` がほとんどの場合 promise の future を完成させる役割をします。そのため必ず出力結果は 1 となるわけです。Promise は、このように非同期の結果を受け取ったり組み合わせたりするためのプレースホルダとしての部品の役割を果たしています。

演習： カウントダウンラッチ

それでは、演習をやってみましょう。Future や Promise の便利な特性を利用して、0 ‖ 1000 ミリ秒間のランダムな時間を待つ 8 個の Future を定義し、そのうちの 3 つが終わり次第すぐにその 3 つの待ち時間を全て出力するという実装をしてみましょう。なお、この動きは、Java の並行処理のためのユーティリティである、CountDownLatch というクラスの動きの一部を模したものとなります。

```
import java.util.concurrent.atomic.AtomicInteger
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.{Promise, Future}
import scala.util.Random

object CountDownLatchSample extends App {
  val indexHolder = new AtomicInteger(0)
  val random = new Random()
  val promises: Seq[Promise[Int]] = for {i <- 1 to 3} yield Promise[Int]
  val futures: Seq[Future[Int]] = for {i <- 1 to 8} yield Future[Int] {
    val waitMilliSec = random.nextInt(1000)
    Thread.sleep(waitMilliSec)
    waitMilliSec
  }
  futures.foreach { f => f.onSuccess {case waitMilliSec =>
    val index = indexHolder.getAndIncrement
    if(index < promises.length) {
      promises(index).success(waitMilliSec)
    }
  }}
  promises.foreach { p => p.future.onSuccess{ case waitMilliSec => println(waitMilliSec)}}
  Thread.sleep(5000)
}
```

上記のコードを簡単に説明すると、指定された処理を行う Future の配列を用意し、それらがそれぞれ成功した時に AtomicInteger で確保されている index をアトミックにインクリメントさせながら、Promise の配列のそれぞれに成功結果を定義しています。そして、最後に Promise の配列から作り出した全ての Future に対して、コンソールに出力をさせる処理を定義します。基本的な Future と Promise を使った処理で表現されていますが、ひとつ気をつけなくてはいけないのは AtomicInteger の部分です。これは Future に渡した関数の中では、同じスレッドが利用されているとは限らないために必要となる部分です。別なスレッドから変更される値に関しては、値を原子的に更新するようにコードを書かなければなりません。プリミティブな値に関して原子的な操作を提供するのが AtomicInteger という Java のクラスとなります。[*2]以上が解答例でした。

ちなみに、このような複雑なイベント処理は既に Java の concurrent パッケージにいくつか実装があるので実際の利用ではそれらを用いることもできます。またもっと複雑なイベントの時間毎の絞込みや合成、分岐などをする際には RxScala というイベントストリームを専門に取り扱うライブラリを利用することができます。この Rx は元々は C#で生まれた Reactive Extensions というライブラリで、現在では様々な言語にポーティングが行われています。

[*2] 値の原始的な更新や同期の必要性などの並行処理に関する様々な話題の詳細な解説は本書の範囲をこえてしまうため割愛します。「Java Concurrency in Practice」ないしその和訳「Java 並行処理プログラミング ーその「基盤」と「最新 API」を究める」や「Effective Java」といった本でこれらの話題について学ぶことが出来ます。

第 18 章

テスト

ソフトウェアをテストすることは多くの開発者が必要なことだと認識していますが、テストという言葉の定義は各人で異なり話が噛み合わない、という状況が多々発生します。このような状況に陥る原因の多くは人や組織、開発するソフトウェアによってコンテキストが異なるにもかかわらず、言葉の定義について合意形成せずに話し始めるためです。

言葉に食い違いがある状態で会話をしても不幸にしかなりません。世の開発者全員で合意することは不可能かもしれませんが、プロジェクト内ではソフトウェアテストという言葉の定義について合意を形成しましょう。また、新しくプロジェクトに配属された場合は定義を確認しておきましょう。

本章では、ソフトウェアテストを下記の定義とします。この定義は古典と呼ばれている書籍『ソフトウェアテストの技法　第二版』をベースにしたものです。

> ソフトウェアテストとは、ソフトウェアが意図されたように動作し意図されないことは全て実行されないように設計されていることを検証するように設計されたプロセス、あるいは一連のプロセスである。

限られたリソースの中でうまくバグを発見できるテストを設計するためには、プロジェクトの仕様、採用した技術、開発の目的を理解する必要があります。こういった知識を得てこそ何を、なぜ、どのように検証するか考えられるようになります。そうして考えられた計画を用いて対象のソフトウェアを検証することが、テストという行為なのです。

テストの分類

テストは幾つかのグループに分類できますが、分類方法についても書籍、組織、チームによってその定義は様々です。プロジェクトに携わる際は、どういった定義でテストを分類しているか確認しておきましょう。参考までに、いくつかの分類例を示します。

- 実践テスト駆動開発での定義
- ユニットテスト
  - オブジェクトは正しく振る舞っているか、またオブジェクトが扱いやすいかどうかをテス

トします。

- インテグレーションテスト
  - 変更できないコードに対して、書いたコードが機能するかテストします。
- 受け入れテスト
  - システム全体が機能するかテストします。
- JSTQB　ソフトウェアテスト標準用語集[*1]での定義
- コンポーネントテスト（component testing）
  - **ユニットテスト**とも呼びます。
  - 個々のソフトウェアコンポーネントのテストを指します。
  - 独立してテストできるソフトウェアの最小単位をコンポーネントと呼びます。
- 統合テスト
  - 統合したコンポーネントやシステムのインタフェースや相互作用の欠陥を摘出するためのテストです。
- システムテスト
  - 統合されたシステムが、特定の要件を満たすことを実証するためのテストのプロセスです。
- 受け入れテスト
  - システムがユーザのニーズ、要件、ビジネスプロセスを満足するかをチェックするためのテストです。
  - このテストによって、システムが受け入れ基準を満たしているか判定したり、ユーザや顧客がシステムを受け入れるかどうかを判定できます。
- 本テキストの初期執筆時に書かれていた定義
- ユニットテスト（Unit Test）
  - **単体テスト**とも呼びます。
  - プログラム全体ではなく小さな単位（例えば関数ごと）に実行されます。このテストの目的はその単体が正しく動作していることを確認することです。ユニットテスト用のフレームワークの種類はとても多く Java の JUnit や PHP の PHPUnit など xUnit と呼ばれることが多いです。
- 結合テスト・統合テスト (Integration Test)
  - プログラム全体が完成してから実際に動作するかを検証するために実行します。人力で行う（例えば機能を開発した本人）ものや Selenium などを使って自動で実行するものがあります。
- システムテスト・品質保証テスト (System Test)
  - 実際に利用される例に則した様々な操作を行い問題ないかを確認します。ウェブアプリケーションの場合ですと、例えばフォームに入力する値の境界値を超えた場合、超えない場合のレスポンスの検証を行ったり様々なウェブブラウザで正常に動作するかなどを確認

[*1] http://www.jstqb.jp/dl/JSTQB-glossary.V2.3.J02.pdf

します。

共通しているのは、分類ごとにテストの目的や粒度が異なること、どのような分類にせよ数多くのテストを通過する必要があることです。

ユニットテスト

ここでは、ユニットテストを**小さな単位で自動実行できるテスト**と定義して、ユニットテストにフォーカスして解説を行います。ユニットテストは Scala でのプログラミングにも密接に関わってくるためです。

ユニットテストを行う理由は大きく 3 つあげられます。

1. 実装の前に満たすべき仕様をユニットテストとして定義し、実装を行うことで要件漏れのない機能を実装することができる
2. ユニットテストによって満たすべき仕様がテストされた状態ならば、安心してリファクタリングすることができる
3. 全ての機能を実装する前に、単体でテストをすることができる

最初の理由であるテストコードをプロダクトコードよりも先に書くことをテストファーストと呼びます。そして、失敗するテストを書きながら実装を進めていく手法のことをテスト駆動開発（TDD: Test Driven Development）といいます。なお、TDD についてはそれ単独で章になり得るので本節では解説しません。

リファクタリングについては、次節で詳しく触れます。

ユニットテストを実装するにあたって、気をつけておきたいことが二点あります。

1. 高速に実行できるテストを実装する
2. 再現性のあるテストを実装する

高速に実行できることでストレスなくユニットテストを実行できるようになります。また、再現性を確保することでバグの原因特定を容易にできます。

リファクタリング

リファクタリングとは、ソフトウェアの仕様を変えること無く、プログラムの構造を扱いやすく変化させることです。

マーチン・ファウラーのリファクタリングでは、リファクタリングを行う理由として 4 つの理由が挙げられています。

1. ソフトウェア設計を向上させるため
2. ソフトウェアを理解しやすくするため

3. バグを見つけやすくするため
4. 早くプログラミングできるようにするため

また同書の中で TDD の提唱者であるケント・ベックは、以下に示す経験則から、リファクタリングは有効であると述べています。

1. 読みにくいプログラムは変更しにくい
2. ロジックが重複しているプログラムは変更しにくい
3. 機能追加に伴い、既存のコード修正が必要になるプログラムは変更しにくい
4. 複雑な条件分岐の多いプログラムは変更しにくい

リファクタリングは中長期的な開発の効率をあげるための良い手段であり、変更が要求されるソフトウェアでは切っても切ることができないプラクティスです。そしてリファクタリングを行う際に、ソフトウェアの仕様を変えなかったことを確認する手段としてユニットテストが必要になるのです。

テスティングフレームワーク

実際にユニットテストのテストコードを書く際には、テスティングフレームワークを利用します。Scala で広く利用されているテスティングフレームワークとして紹介されるのは以下の 2 つです。

- Specs2
- ScalaTest

今回は、マクロを用いて実装されている power assert という便利な機能を使いたいため、ScalaTest を利用します[*2]。

ScalaTest は、テスティングフレームワークの中でも 振舞駆動開発（BDD :Behavior Driven Development）をサポートしているフレームワークです。BDD では、テスト内にそのプログラムに与えられた機能的な外部仕様を記述させることで、テストが本質的に何をテストしようとしているのかをわかりやすくする手法となります。基本この書き方に沿って書いていきます。

テストができる sbt プロジェクトの作成

では、実際にユニットテストを書いてみましょう。まずはプロジェクトを作成します。

適当な作業フォルダにて以下を実行します。ここでは、`scalatest_study` を作り、さらに中に `src/main/scala` と `src/test/scala` の 2 つのフォルダを作りましょう。

`build.sbt` を用意して、以下を記述しておきます。

[*2] 渡された条件式の実行過程をダイアグラムで表示する `assert` は、一般に “power assert” と呼ばれています

```
name := "scalatest_study"

version := "1.0"

scalaVersion := "2.11.8"

libraryDependencies += "org.scalatest" %% "scalatest" % "2.2.6" % "test"
```

その後、scalatest_study フォルダ内で、sbt compile を実行してみましょう。

```
[info] Set current project to scalatest_study (in build file:/Users/dwango/workspace/scalatest_st
[info] Updating {file:/Users/dwango/workspace/scalatest_study/scalatest_study/}scalatest_study...
[info] Resolving jline#jline;2.12.1 ...
[info] downloading http://repo1.maven.org/maven2/org/scalatest/scalatest_2.11/2.2.6/scalatest_2.1
[info]  [SUCCESSFUL ] org.scalatest#scalatest_2.11;2.2.6!scalatest_2.11.jar(bundle) (10199ms)
[info] Done updating.
[success] Total time: 11 s, completed 2015/04/09 16:48:42
```

以上のように表示されれば、これで準備は完了です。

Calc クラスとそのテストを実際に作る

それでは、具体的なテストを実装してみましょう。下記の仕様のを満たす Calc クラスを作成し、それらをテストしていきます。

- 整数の配列を取得し、それらを足し合わせた整数を返すことができる sum 関数を持つ
- 整数を 2 つ受け取り、分子を分母で割った浮動小数点の値を返すことができる div 関数を持つ
- 整数値を 1 つ受け取り、その値が素数であるかどうかのブール値を返す isPrime 関数を持つ

これを実装した場合、src/main/scala/Calc.scala は以下のようになります。

```
class Calc {

  /** 整数の配列を取得し、それらを出し合わせた整数を返す
    *
    * Int の最大を上回った際にはオーバーフローする
    */
  def sum(seq: Seq[Int]): Int = seq.foldLeft(0)(_ + _)

  /** 整数を 2 つ受け取り、分子を分母で割った浮動小数点の値を返す
    *
    * 0 で割ろうとした際には実行時例外が投げられる
    */
  def div(numerator: Int, denominator: Int): Double = {
    if (denominator == 0) throw new ArithmeticException("/ by zero")
    numerator.toDouble / denominator.toDouble
```

```
  }

  /** 整数値を一つ受け取り、その値が素数であるかどうかのブール値を返す */
  def isPrime(n: Int): Boolean = {
    if (n < 2) false else !((2 to Math.sqrt(n).toInt) exists (n % _ == 0))
  }
}
```

次にテストケースについて考えます。

- sum 関数
  - 整数の配列を取得し、それらを足し合わせた整数を返すことができる
  - Int の最大を上回った際にはオーバーフローする
- div 関数
  - 整数を 2 つ受け取り、分子を分母で割った浮動小数点の値を返す
  - 0 で割ろうとした際には実行時例外が投げられる
- isPrime 関数
  - その値が素数であるかどうかのブール値を返す
  - 100 万以下の値の素数判定を一秒以内で処理できる

以上のようにテストを行います。 基本的にテストの設計は、

1. 機能を満たすことをテストする
2. 機能が実行できる境界値に対してテストする
3. 例外やログがちゃんと出ることをテストする

以上の考えが重要です。

XP（エクストリームプログラミング）のプラクティスに、不安なところを徹底的にテストするという考えがあり、基本それに沿います。

ひとつ目の満たすべき機能が当たり前に動くは当たり前のこととして、2 つ目に不安な要素のある境界値をしっかりテストする、というのはテストするケースを減らし、テストの正確性をあげるためのプラクティスの境界値テストとしても知られています。そして最後は、レアな事象ではあるけれど動かないと致命的な事象の取り逃しにつながる例外やログについてテストも非常に重要なテストです。このような例外やログにテストがないと例えば 1 か月に 1 度しか起こらないような不具合に対する対処が原因究明できなかったりと大きな問題につながってしまいます。

最小のテストを書いてみます。`src/test/scala/CalcSpec.scala` を以下のように記述します。

```
import org.scalatest._

class CalcSpec extends FlatSpec with DiagrammedAssertions {

  val calc = new Calc
```

```
  "sum 関数" should "整数の配列を取得し、それらを足し合わせた整数を返すことができる" in {
    assert(calc.sum(Seq(1, 2, 3)) === 6)
    assert(calc.sum(Seq(0)) === 0)
    assert(calc.sum(Seq(-1, 1)) === 0)
    assert(calc.sum(Seq()) === 0)
  }

  it should "Int の最大を上回った際にはオーバーフローする" in {
    assert(calc.sum(Seq(Integer.MAX_VALUE, 1)) === Integer.MIN_VALUE)
  }
}
```

テストクラスに DiagrammedAssertions をミックスインし、assert メソッドの引数に期待する条件を記述していきます[*3]。DiagrammedAssertions を使うことで、覚えるべき API を減らしつつテスト失敗時に多くの情報を表示できるようになります。

テストを実装したら sbt test でテストを実行してください。以下のような実行結果が表示されます。

```
[info] Loading project definition from /Users/dwango/workspace/scalatest_study/project
[info] Set current project to scalatest_study (in build file:/Users/dwango/workspace/scalatest_st
[info] Compiling 1 Scala source to /Users/dwango/workspace/scalatest_study/target/scala-2.11/clas
[info] Compiling 1 Scala source to /Users/dwango/workspace/scalatest_study/target/scala-2.11/test
[info] CalcSpec:
[info] sum 関数
[info] - should 整数の配列を取得し、それらを足し合わせた整数を返すことができる
[info] - should Int の最大を上回った際にはオーバーフローする
[info] Run completed in 570 milliseconds.
[info] Total number of tests run: 2
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 2, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[success] Total time: 12 s, completed 2015/12/25 1:25:56
```

実行結果から、すべてのテストに成功したことを確認できます。なお、わざと失敗した場合にはどのように表示されるのか確認してみましょう。

```
[info] Loading project definition from /Users/dwango/workspace/scalatest_study/project
[info] Set current project to scalatest_study (in build file:/Users/dwango/workspace/scalatest_st
[info] Compiling 1 Scala source to /Users/dwango/workspace/scalatest_study/target/scala-2.11/test
[info] CalcSpec:
```

[*3] Scala には Predef にも assert が存在しますが、基本的に使うことはありません

```
[info] sum 関数
[info] - should 整数の配列を取得し、それらを足し合わせた整数を返すことができる *** FAILED ***
[info]   assert(calc.sum(Seq(1, 2, 3)) === 7)
[info]          |    |  ||   |  |  |   |   |
[info]          |    6  ||   1  2  3   |   7
[info]          |       |List(1, 2, 3) false
[info]          |       6
[info]          Calc@e72a964 (CalcSpec.scala:8)
[info] - should Int の最大を上回った際にはオーバーフローする
[info] Run completed in 288 milliseconds.
[info] Total number of tests run: 2
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 1, failed 1, canceled 0, ignored 0, pending 0
[info] *** 1 TEST FAILED ***
[error] Failed tests:
[error]     CalcSpec
[error] (test:test) sbt.TestsFailedException: Tests unsuccessful
[error] Total time: 7 s, completed 2015/12/25 1:39:59
```

どこがどのように間違ったのかを指摘してくれます。

次に、例外が発生をテストする場合について記述してみましょう。div 関数までテストの実装を進めます。

```
import org.scalatest._

class CalcSpec extends FlatSpec with DiagrammedAssertions {

  val calc = new Calc

  // ...

  "div 関数" should "整数を 2 つ受け取り、分子を分母で割った浮動小数点の値を返す" in {
    assert(calc.div(6, 3) === 2.0)
    assert(calc.div(1, 3) === 0.3333333333333333)
  }

  it should "0 で割ろうとした際には実行時例外が投げられる" in {
    intercept[ArithmeticException] {
      calc.div(1, 0)
    }
  }
}
```

上記では最後の部分でゼロ除算の際に投げられる例外をテストしています。intercept[Exception] という構文で作ったスコープ内で投げられる例外がある場合には成功となり、例外がない場合には逆にテストが失敗します。

最後にパフォーマンスを保証するテストを書きます。なお、本来ユニットテストは時間がかかるテストを書くべきではありませんが、できるだけ短い時間でそれを判定できるように実装します。

```
import org.scalatest._
import org.scalatest.concurrent.Timeouts
import org.scalatest.time.SpanSugar._

class CalcSpec extends FlatSpec with DiagrammedAssertions with Timeouts {

  val calc = new Calc

  // ...

  "isPrime 関数" should "その値が素数であるかどうかのブール値を返す" in {
    assert(calc.isPrime(0) === false)
    assert(calc.isPrime(-1) === false)
    assert(calc.isPrime(2))
    assert(calc.isPrime(17))
  }

  it should "100 万以下の値の素数判定を一秒以内で処理できる" in {
    failAfter(1000 millis) {
      assert(calc.isPrime(9999991))
    }
  }
}
```

Timeouts というトレイトを利用することで failAfter という処理時間をテストする機能を利用できるようになります。

最終的に全てのテストをまとめて sbt test で実行すると以下の様な出力が得られます。

```
[info] Loading project definition from /Users/dwango/workspace/scalatest_study/project
[info] Set current project to scalatest_study (in build file:/Users/dwango/workspace/scalatest_st
[info] Compiling 1 Scala source to /Users/dwango/workspace/scalatest_study/target/scala-2.11/test
[info] CalcSpec:
[info] sum 関数
[info] - should 整数の配列を取得し、それらを足し合わせた整数を返すことができる
[info] - should Int の最大を上回った際にはオーバーフローする
[info] div 関数
[info] - should 整数を 2 つ受け取り、分子を分母で割った浮動小数点の値を返す
[info] - should 0 で割ろうとした際には実行時例外が投げられる
[info] isPrime 関数
```

```
[info] - should その値が素数であるかどうかのブール値を返す
[info] - should 100 万以下の値の素数判定を一秒以内で処理できる
[info] Run completed in 280 milliseconds.
[info] Total number of tests run: 6
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 6, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[success] Total time: 8 s, completed 2015/12/25 1:43:22
```

以上が基本的なテストを実装するための機能の紹介でした。BDD でテストを書くことによってテストによってどのような仕様が満たされた状態であるのかというのがわかりやすい状況になっていることがわかります。

モック

モックとは、テストをする際に必要となるオブジェクトを偽装して用意できる機能です。以下の様なモックライブラリが存在しています。

- ScalaMock
- EasyMock
- JMock
- Mockito

ここでは、ドワンゴ社内で利用率の高い Mockito を利用してみましょう。`build.sbt` に以下を追記することで利用可能になります。

```
libraryDependencies += "org.mockito" %% "mockito-core" % "1.10.19" % "test"
```

せっかくなので、先ほど用意した Calc クラスのモックを用意して、モックに sum の振る舞いを仕込んで見ましょう。

```
import org.scalatest.{FlatSpec, DiagrammedAssertions}
import org.scalatest.concurrent.Timeouts
import org.scalatest.mock.MockitoSugar
import org.mockito.Mockito._

class CalcSpec extends FlatSpec with DiagrammedAssertions with Timeouts with MockitoSugar {

  // ...

  "Calc のモックオブジェクト" should "振る舞いを偽装することができる" in {
    val mockCalc = mock[Calc]
    when(mockCalc.sum(Seq(3, 4, 5))).thenReturn(12)
```

```
    assert(mockCalc.sum(Seq(3, 4, 5)) === 12)
  }
}
```

MockitoSuger というトレイトをミックスインすることで、ScalaTest 独自の省略記法を用いて Mockito を利用できるようになります。val mockCalc = mock[Calc] でモックオブジェクトを作成し、when(mockCalc.sum(Seq(3, 4, 5)).thenReturn(12) で振る舞いを作成しています。

そして最後に、assert(mockCalc.sum(Seq(3, 4, 5)) === 12) でモックに仕込んだ偽装された振る舞いをテストしています。

以上のようなモックの機能は、実際には時間がかかってしまう通信などの部分を高速に動かすために利用されています。

モックを含め、テストの対象が依存しているオブジェクトを置き換える代用品の総称をテストダブル[*4]と呼びます。

コードカバレッジの測定

テストを行った際に、テストが機能のどれぐらいを網羅できているのかを知る方法として、コードカバレッジを図るという方法があります。ここでは、scoverage を利用します。

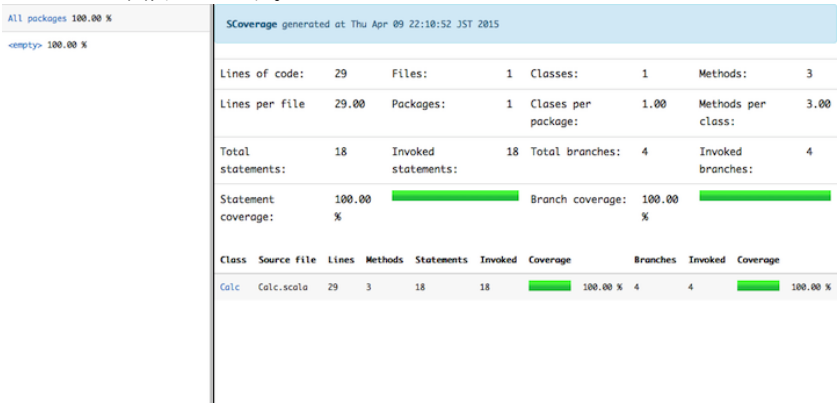
過去、SCCT というプロダクトがあったのですが紆余曲折あり、今はあまりメンテンナンスされていません。

project/plugins.sbt に以下のコードを記述します。

```
resolvers += Classpaths.sbtPluginReleases

addSbtPlugin("org.scoverage" % "sbt-scoverage" % "1.3.3")
```

その後、sbt clean coverage test を実行することで、target/scala-2.11/scoverage-report/index.html にレポートが出力されます。



以上の出力から、今回のテストはカバレッジ 100 %であることがわかります。

[*4] モック以外の仕組みについては xUnit Test Patterns を参照してください

ここで、カバレッジを計測することについて注意点を述べておきます。

得られたカバレッジはあくまで" そのカバレッジツールによって得られた数値" でしかありません。数値自体が絶対的な評価を表しているわけではなく、書かれたプロダクトコードに対して、テストコードが N %カバーしているという事実を示しているだけです。カバレッジはツールや測定手法によって結果が変動します。例えば JavaScript 用のカバレッジツールである istanbul は total が 0 件の場合 100% と表示しますが、これは単にツールの実装がそうなっているというだけの話です。

利用しているカバレッジツールがどのように動作するのか把握した上で、カバレッジレポートからどのコードが実行されなかったのか情報収集しましょう。数値だけに目を向けるのではなく、カバレッジ測定によってどういった情報が得られたのかを考えてください。そうすれば、ツールに振り回されることなくテストの少ない部分を発見できるでしょう。

コードスタイルチェック

なおテストとは直接は関係ありませんが、ここまでで紹介したテストは、実際には Jenkins などの継続的インテグレーションツール（CI ツール）で実施され、リグレッションを検出するためにつかわれます。その際に、CI の一環として一緒に行われることが多いのがコードスタイルチェックです。

ここでは、ScalaStyle を利用します。

使い方は、`project/plugins.sbt` に以下のコードを記述します。

```
addSbtPlugin("org.scalastyle" %% "scalastyle-sbt-plugin" % "0.6.0")
```

その後、`sbt scalastyleGenerateConfig` を一度だけ実施後、`sbt scalastyle` を実行します。

実行すると、下記のように警告が表示されます。

```
[info] Loading project definition from /Users/dwango/workspace/scalatest_study/project
[info] Set current project to scalatest_study (in build file:/Users/dwango/workspace/scalatest_st
[info] scalastyle using config /Users/dwango/workspace/scalatest_study/scalastyle-config.xml
[warn] /Users/dwango/workspace/scalatest_study/src/main/scala/Calc.scala:1: Header does not match
[info] Processed 1 file(s)
[info] Found 0 errors
[info] Found 1 warnings
[info] Found 0 infos
[info] Finished in 12 ms
[success] created output: /Users/dwango/workspace/scalatest_study/target
[success] Total time: 1 s, completed 2015/04/09 22:17:40
```

これはヘッダに特定のテキストが入っていないというルールに対して警告を出してくれているもので、`scalastyle-config.xml` でルール変更を行うことができます。なおデフォルトの設定では、Apache ライセンスの記述を入れなくては警告を出す設定になっています。これらもテスティングフ

レームワークやカバレッジ測定と同時に導入、設定してしまいましょう。

テストを書こう

開発者として意識してほしいことを挙げておきます。

- 不具合を見つけたら、可能であれば再現手順もあわせて報告する
- パッチを送る際はテストを含める
- 不具合にテストを書いて立ち向かう

他人から報告された不具合は、伝聞ということもありどうしても再現させづらい場合があります。その結果として「私の環境では再現しませんでした」と言われて終わってしまうのでは、不具合を修正するせっかくのチャンスを逃すことになってしまいます。そんな状況を回避するためにも、可能であれば不具合の再現手順や再現コードを用意し、不具合報告に含めましょう。再現手順を元に素早く不具合が修正される可能性が高まります。

他人のコードをテストするのは、ちょっとしたコードであっても骨の折れる作業です。そんな作業を、限られたリソースですべてのパッチに対して行うのは不可能に近いので、プロジェクトによってはテストコードのないパッチはレビュー対象から外すことが多いです。パッチを送る場合は相手側が自信を持ってパッチを取り込めるよう、テストを含めておきましょう。

第 19 章

Java との相互運用（★★★）

Scala と Java

Scala は JVM(Java Virtual Machine) の上で動作するため、Java のライブラリのほとんどをそのまま Scala から呼びだすことができます。また、現状では、Scala の標準ライブラリだけでは、どうしても必要な機能が足りず、Java の機能を利用せざるを得ないことがあります。ただし、Java の機能と言っても、Scala のそれとほとんど同じように利用することができます。

import

Java のライブラリを import するためには、Scala でほとんど同様のことを記述すれば OK です。

```
import java.util.*;
import java.util.ArrayList;
```

は

```
import java.util._
import java.util.ArrayList
```

と同じ意味になります。注意するべきは、Java でのワイルドカードインポートが、*ではなく_になった程度です。

インスタンスの生成

インスタンスの生成も Java と同様にできます。Java での

```
ArrayList<String> list = new ArrayList<>();
```

というコードは Scala では

```
scala> val list = new ArrayList[String]()
list: java.util.ArrayList[String] = []
```

と記述することができます。

練習問題

java.util.HashSet クラスのインスタンスを new を使って生成してみましょう。

```
scala> import java.util.HashSet
import java.util.HashSet

scala> val set = new HashSet[String]
set: java.util.HashSet[String] = []
```

インスタンスメソッドの呼び出し

インスタンスメソッドの呼び出しも同様です。

```
list.add("Hello");
list.add("World");
```

は

```
scala> list.add("Hello")
res0: Boolean = true

scala> list.add("World")
res1: Boolean = true
```

と同じです。

練習問題

java.lang.System クラスのフィールド out のインスタンスメソッド println を引数 "Hello, World!" として呼びだしてみましょう。

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

static メソッドの呼び出し

static メソッドの呼び出しも Java の場合とほとんど同様にできますが、1 つ注意点があります。それは、Scala では static メソッドは継承されない（というより static メソッドという概念がない）ということです。これは、クラス A が static メソッド foo を持っていたとして、A を継承した B に対して B.foo() とすることはできず、A.foo() としなければならないという事を意味します。それ以外の点については Java の場合とほぼ同じです。

現在時刻をミリ秒単位で取得する System.currentTimeMillis() を Scala から呼び出してみましょう。

```
scala> System.currentTimeMillis()
res0: Long = 1416357548906
```

表示される値はみなさんのマシンにおける時刻に合わせて変わりますが、問題なく呼び出せているはずです。

■練習問題　java.lang.System クラスの static メソッド exit() を引数 0 として呼びだしてみましょう。どのような結果になるでしょうか。

```
System.exit(0)
```

実行中の Scala プログラム（プロセス）が終了する。

static フィールドの参照

static フィールドの参照も Java の場合と基本的に同じですが、static メソッドの場合と同じ注意点が当てはまります。つまり、static フィールドは継承されない、ということです。たとえば、Java では JFrame.EXIT_ON_CLOSE が継承されることを利用して、

```
import javax.swing.JFrame;

public class MyFrame extends JFrame {
  public MyFrame() {
    setDefaultCloseOperation(EXIT_ON_CLOSE); //JFrame を継承しているので、EXIT_ON_CLOSE だけで OK
  }
}
```

のようなコードを書くことができますが、Scala では同じように書くことができず、

```
scala> import javax.swing.JFrame

class MyFrame extends JFrame {
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE) //JFrame. を明示しなければならない
}
```

のように書く必要があります。

現実のプログラミングでは、Scala の標準ライブラリだけでは必要なライブラリが不足している場面に多々遭遇しますが、そういう場合は既にあるサードパーティの Scala ライブラリか Java ライブラリを直接呼びだすのが基本になります。

■練習問題　Scala で Java の static フィールドを参照しなければならない局面を 1 つ以上挙げてみましょう。

java.lang.System クラスの static フィールド err を参照する場合

Scala の型と Java の型のマッピング

Java の型は適切に Scala にマッピングされます。たとえば、`System.currentTimeMillis()` が返す型は long 型ですが、Scala の標準の型である `scala.Long` にマッピングされます。Scala の型と Java の型のマッピングは次のようになります。

Java のすべてのプリミティブ型に対応する Scala の型が用意されていることがわかりますね！
また、`java.lang` パッケージにあるクラスは全て Scala から import 無しに使えます。

また、参照型についても Java 同様にクラス階層の中に組み込まれています。たとえば、Java で言う `int[]` は `Array[Int]` と書きますが、これは `AnyRef` のサブクラスです。ということは、Scala で `AnyRef` と書くことで `Array[Int]` を `AnyRef` 型の変数に代入可能です。ユーザが定義したクラスも同様で、基本的に `AnyRef` を継承していることになっています。（ただし、value class というものがあり、それを使った場合は少し事情が異なりますがここでは詳細には触れません）

null と Option

Scala の世界では null を使うことはなく、代わりに Option 型を使います。一方で、Java のメソッドを呼び出したりすると、返り値として null が返ってくることがあります。Scala の世界ではできるだけ null を取り扱いたくないのでこれは少し困ったことです。幸いにも、Scala では `Option(value)` とすることで、`value` が null のときは `None` が、null でないときは `Some(value)` を返すようにできます。

java.util.Map を使って確かめてみましょう。

```
scala> val map = new java.util.HashMap[String, Int]()
map: java.util.HashMap[String,Int] = {}

scala> map.put("A", 1)
res3: Int = 0

scala> map.put("B", 2)
res4: Int = 0

scala> map.put("C", 3)
res5: Int = 0

scala> Option(map.get("A"))
res6: Option[Int] = Some(1)

scala> Option(map.get("B"))
res7: Option[Int] = Some(2)

scala> Option(map.get("C"))
res8: Option[Int] = Some(3)

scala> Option(map.get("D"))
res9: Option[Int] = None
```

ちゃんと null が Option にラップされていることがわかります。Scala の世界から Java のメソッドを呼びだすときは、返り値をできるだけ Option() でくるむように意識しましょう。

JavaConverters

Java のコレクションと Scala のコレクションはインタフェースに互換性がありません。これでは、Scala のコレクションを Java のコレクションに渡したり、逆に返ってきた Java のコレクションを Scala のコレクションに変換したい場合に不便です。そのような場合に便利なのが JavaConverters です。使い方はいたって簡単で、

```
import scala.collection.JavaConverters._
```

とするだけです。これで、Java と Scala のコレクションのそれぞれに asJava() や asScala() といったメソッドが追加されるのでそのメソッドを以下のように呼び出せば良いです。

```
scala> import scala.collection.JavaConverters._
import scala.collection.JavaConverters._

scala> import java.util.ArrayList
import java.util.ArrayList

scala> val list = new ArrayList[String]()
list: java.util.ArrayList[String] = []

scala> list.add("A")
res10: Boolean = true

scala> list.add("B")
res11: Boolean = true

scala> val scalaList = list.asScala
scalaList: scala.collection.mutable.Buffer[String] = Buffer(A, B)
```

Buffer は Scala の変更可能なリストのスーパークラスですが、ともあれ、asScala メソッドによって Java のコレクションを Scala のそれに変換することができていることがわかります。そのほかのコレクションについても同様に変換できますが、詳しくは API ドキュメントを参照してください。

■練習問題　`scala.collection.mutable.ArrayBuffer` 型の値を生成してから、JavaConverters を使って java.util.List 型に変換してみましょう。なお、`ArrayBuffer` には 1 つ以上の要素を入れておくこととします。

```
scala> import scala.collection.mutable.ArrayBuffer
import scala.collection.mutable.ArrayBuffer

scala> import scala.collection.JavaConverters._
import scala.collection.JavaConverters._
```

```
scala> val buffer = new ArrayBuffer[String]
buffer: scala.collection.mutable.ArrayBuffer[String] = ArrayBuffer()

scala> buffer += "A"
res12: buffer.type = ArrayBuffer(A)

scala> buffer += "B"
res13: buffer.type = ArrayBuffer(A, B)

scala> buffer += "C"
res14: buffer.type = ArrayBuffer(A, B, C)

scala> val list = buffer.asJava
list: java.util.List[String] = [A, B, C]
```

第 20 章

S99 の案内（★）

ここでは、Scala を用いたプログラミングについてより深く理解するために適していると思われる問題集である S99 について簡単に紹介します。

S-99: Ninety-Nine Scala Problems

URL: http://aperiodic.net/phil/scala/s-99/
元々は Prolog 用の Ninety-Nine Prolog Problems を Scala 版に移植した問題集です。

- P01 ‖ P28：リスト操作
- P31 ‖ P41：数値演算等
- P46 ‖ P50：論理演算
- P55 ‖ P69：二分木
- P70 ‖ P73：多分木
- P80 ‖ P89：グラフ
- P90 ‖ P99：その他の問題

と問題の種類に応じて区分けされています。とりわけ、Scala を使うときはコレクションやリスト操作を多用するので、P01 ‖ P28 については一通り解けるようになっておくと良いでしょう。その他は個人の興味に応じて適宜色々な問題を解いてみてください。
なお、S99 の問題の模範解答として、
https://github.com/dwango/S99
というリポジトリを用意してあります。全ての解答が整備されているわけではありませんが、考えても解答がわからない場合は参考にしても良いでしょう。また、各問題の解答についてのテストケースも書いてあります。

第 21 章

トレイトの応用編：依存性の注入によるリファクタリング

ここではトレイトの応用編として、大きなクラスをリファクタリングする過程を通してトレイトを実際どのように使っていくかを学んでいきましょう。さらに大規模システム開発で使われる依存性の注入という技法についても紹介します。

サンプルプログラム

今回使われるサンプルプログラムは以下の場所にあります。実際に動かしたい場合は個々のディレクトリに入り、sbt を起動してください。

リファクタリング前のプログラム

リファクタリング後のプログラム

リファクタリング前のプログラムの紹介

今回は今までより実践的なプログラムを考えてみましょう。ユーザーの登録と認証を管理する `UserService` というクラスです。

include

include

include

`UserService` は以下のような機能があります。

- `register` メソッドはユーザーの登録をおこなうメソッドで、ユーザーの名前とパスワードを引数として受け取り、名前の最大長のチェックと既に名前が登録されているかどうかを調べて、ストレージに保存する
- `login` メソッドはユーザーの認証をおこなうメソッドで、ユーザーの名前とパスワードを受け取り、ストレージに保存されているユーザーの中から同名のユーザーを見つけ出し、パスワー

ドをチェックする

- この他のストレージ機能とパスワード機能のメソッドは内部的に使われるのみである

上記のプログラムでは実際に動かすことができるように ScalikeJDBC というデータベース用のライブラリと jBCrypt というパスワースのハッシュ値を計算するライブラリが使われていますが、実装の詳細を理解する必要はありません。既にメソッドの実装を説明したことがある場合は実装を省略し???で書くことがあります。

リファクタリング：公開する機能を制限する

さて、モジュール化という観点で UserService を見ると、どういった問題が考えられるでしょうか？

1 つは UserService が必要以上に情報を公開しているということです。UserService の役割は register メソッドを使ったユーザー登録と login メソッドを使ったユーザーの認証です。しかしストレージ機能である insert メソッドや find メソッドも公開してしまっています。

register メソッドはユーザーの名前の最大長や既にユーザーが登録されているかどうかチェックしていますが、insert メソッドはそういったチェックをせずにデータベースに保存しています。もし insert メソッドを直接使われた場合、想定外に名前が長いユーザーや、名前が重複したユーザーが保存されてしまいます。

find メソッドも同様の問題があります。login メソッドではなく直接 find メソッドが使われた場合、パスワードのチェックなしにユーザーの情報を取得することができます。

では、どのような修正が考えられるでしょうか。まず考えられるのは外部から使ってほしくないメソッドを private にすることです。

```
class UserService {
  // メソッドの実装は同じなので???で代用しています
  val maxNameLength = 32

  // ストレージ機能
  private[this] def insert(user: User): User = ???

  private[this] def createUser(rs: WrappedResultSet): User = ???

  private[this] def find(name: String): Option[User] = ???

  private[this] def find(id: Long): Option[User] = ???

  // パスワード機能
  private[this] def hashPassword(rawPassword: String): String = ???

  private[this] def checkPassword(rawPassword: String, hashedPassword: String): Boolean = ???

  // ユーザー登録
```

```
  def register(name: String, rawPassword: String): User = ???

  // ユーザー認証
  def login(name: String, rawPassword: String): User = ???
}
```

これで insert メソッドや find メソッドは外部から呼びだすことができなくなったので、先ほどの問題は起きなくなりました。

しかしトレイトを使って同じように公開したい機能を制限することもできます。まず、公開するメソッドだけを集めた新しいトレイト UserService を作ります。

```
trait UserService {
  val maxNameLength = 32

  def register(name: String, rawPassword: String): User

  def login(name: String, rawPassword: String): User
}
```

そして、このトレイトの実装クラス UserServiceImpl を作ります。

```
class UserServiceImpl extends UserService {
  // メソッドの実装は同じなので???で代用しています

  // ストレージ機能
  def insert(user: User): User = ???

  def createUser(rs: WrappedResultSet): User = ???

  def find(name: String): Option[User] = ???

  def find(id: Long): Option[User] = ???

  // パスワード機能
  def hashPassword(rawPassword: String): String = ???

  def checkPassword(rawPassword: String, hashedPassword: String): Boolean = ???

  // ユーザー登録
  def register(name: String, rawPassword: String): User = ???

  // ユーザー認証
  def login(name: String, rawPassword: String): User = ???
}
```

UserService の利用者は実装クラスではなく公開トレイトのほうだけを参照するようにすれば、チェックされていないメソッドを使って不整合データができてしまう問題も起きません。

このようにモジュールのインタフェースを定義し、公開する機能を制限するのもトレイトの使われ方の 1 つです。Java のインタフェースと同じような使い方ですね。

リファクタリング：大きなモジュールを分割する

次にこのモジュールの問題点として挙げられるのは、モジュールが多くの機能を持ちすぎているということです。UserService はユーザー登録とユーザー認証をおこなうサービスですが、付随してパスワードをハッシュ化したり、パスワードをチェックする機能も持っています。

このパスワード機能は UserService 以外でも使いたいケースが出てくるかもしれません。たとえばユーザーが重要な操作をした場合に再度パスワードを入力させ、チェックしたい場合などです。

このような場合、1 つのモジュールを複数のモジュールに分割することが考えられます。あたらしくパスワード機能だけを持つ PasswordService と PasswordServiceImpl を作ってみます。

include

そして、先ほど作った UserServiceImpl に PasswordServiceImpl を継承して使うようにします。

```
class UserServiceImpl extends UserService with PasswordServiceImpl {
  // メソッドの実装は同じなので???で代用しています

  // ストレージ機能
  def insert(user: User): User = ???

  def createUser(rs: WrappedResultSet): User = ???

  def find(name: String): Option[User] = ???

  def find(id: Long): Option[User] = ???

  // ユーザー登録
  def register(name: String, rawPassword: String): User = ???

  // ユーザー認証
  def login(name: String, rawPassword: String): User = ???
}
```

これでパスワード機能を分離することができました。分離したパスワード機能は別の用途で使うこともできるでしょう。

同じようにストレージ機能も UserRepository として分離してみます。

include

すると、UserServiceImpl は以下のようになります。

```
class UserServiceImpl extends UserService with PasswordServiceImpl with UserRepositoryImpl {
  // メソッドの実装は同じなので???で代用しています

  // ユーザー登録
  def register(name: String, rawPassword: String): User = ???

  // ユーザー認証
  def login(name: String, rawPassword: String): User = ???
```

```
}
```

これで大きな UserService モジュールを複数の機能に分割することできました。

依存性の注入によるリファクタリング

さて、いよいよこの節の主題である依存性の注入によるリファクタリングの話に入りましょう。

ここまでトレイトを使って公開する機能を定義し、モジュールを分割し、リファクタリングを進めてきましたが、UserService にはもう 1 つ大きな問題があります。モジュール間の依存関係が分離できていないことです。

たとえば UserServiceImpl のユニットテストをすることを考えてみましょう。ユニットテストは外部のシステムを使わないので、ローカル環境でテストしやすく、並行して複数のテストをすることもできます。また、単体の機能のみをテストするため失敗した場合、問題の箇所がわかりやすいという特徴もあります。

UserServiceImpl は UserRepositoryImpl に依存しています。UserRepositoryImpl は ScalikeJDBC を使った外部システムであるデータベースのアクセスコードがあります。このままの UserServiceImpl でユニットテストを作成した場合、テストを実行するのにデータベースを用意しなければならず、データベースを共用する場合複数のテストを同時に実行するのが難しくなります。さらにテストに失敗した場合 UserServiceImpl に原因があるのか、もしくはデータベースの設定やテーブルに原因があるのか、調査しなければなりません。これではユニットテストとは言えません。

そこで具体的な UserRepositoryImpl への依存を分離することが考えられます。このために使われるのが依存性の注入と呼ばれる手法です。

依存性の注入とは？

まずは一般的な「依存性の注入（Dependency Injection、DI）」の定義を確認しましょう。

依存性の注入について Wikipedia の Dependency injection の項目を見てみますと、

- Dependency とは実際にサービスなどで使われるオブジェクトである
- Injection とは Dependency を使うオブジェクトに渡すことである

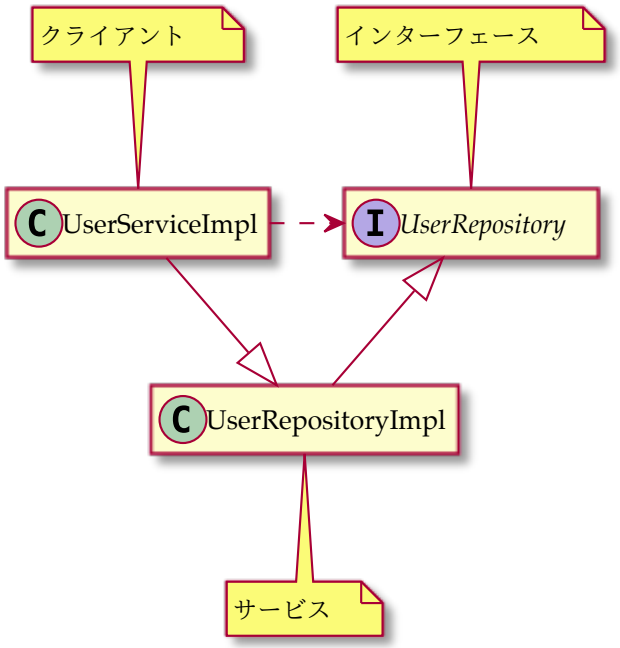
とあります。さらに DI には以下の 4 つの役割が登場するとあります。

- 使われる対象の「サービス」
- サービスを使う（依存する）「クライアント」
- クライアントがどうサービスを使うかを定めた「インタフェース」
- サービスを構築し、クライアントに渡す「インジェクタ」

これらの役割について、今回の例の UserRepository、UserRepositoryImpl、UserServiceImpl で考えてみます。Wikipedia 中の「サービス」という用語と、これまでの例の中で登場するサービス

という言葉は別の意味なので注意してください。

まずは DI を使っていない状態のクラス図を見てみましょう。

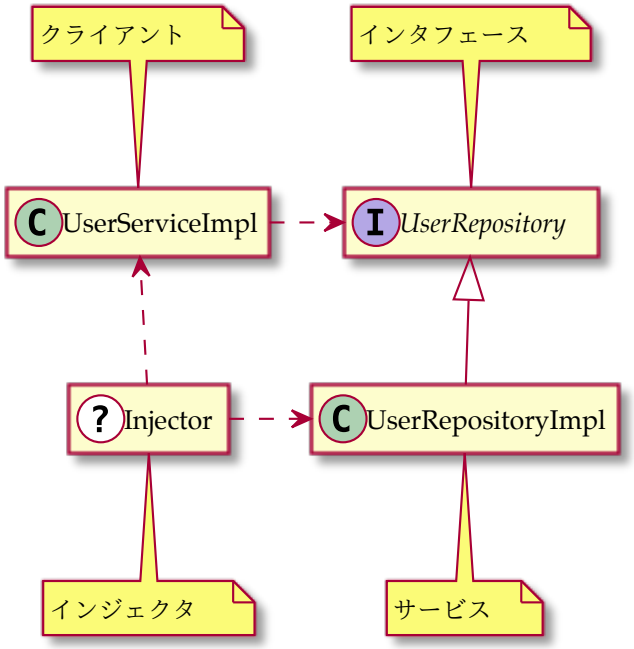


このクラス図の役割を表にしてみます。

DI の役割	コード上の名前	説明
インタフェース	UserRepository	抽象的なインタフェース
サービス	UserRepositoryImpl	具体的な実装
クライアント	UserServiceImpl	UserRepository の利用者

DI を使わない状態では `UserRepository` というインタフェースが定義されているのにもかかわらず、`UserServiceImpl` は `UserRepositoryImpl` を継承することで実装も参照していました。これではせっかくインタフェースを分離した意味がありません。`UserServiceImpl` が `UserRepository` インタフェースだけを参照（依存）するようにすれば、具体的な実装である `UserRepositoryImpl` の変更に影響されることはありません。この問題を解決するのが DI の目的です。

それでは DI のインジェクタを加えて、上記のクラス図を修正しましょう。



謎のインジェクタの登場により `UserServiceImpl` から `UserRepositoryImpl` への参照がなくなりました。おそらくインジェクタは何らかの手段でサービスである `UserRepositoryImpl`（Dependency）をクライアントである `UserServiceImpl` に渡しています（Injection）。このインジェクタの動作を指して「Dependency Injection」と呼ぶわけです。そして、このインジェクタをどうやって実現するか、それが DI 技術の核心に当たります。

依存性の注入の利点

では、この依存性の注入を使うとどのような利点があるのでしょうか。

1 つはクライアントがインタフェースだけを参照することにより、具体的な実装への参照が少なくなり**コンポーネント同士が疎結合になる**という点が挙げられます。たとえば `UserRepositoryImpl` のクラス名やパッケージ名が変更されても `UserServiceImpl` には何の影響もなくなります。

次に挙げられる点は具体的な実装を差し替えることにより**クライアントの動作がカスタマイズ可能**になるという点です。たとえば今回の例では `UserRepositoryImpl` は ScalikeJDBC の実装でしたが、MongoDB に保存する `MongoUserRepositoryImpl` を新しく作って `UserServiceImpl` に渡せばクライアントを MongoDB に保存するように変更することができます。

また DI は設計レベルでも意味があります。 DI を使うと**依存関係逆転の原則を実現できます**。通常の手続き型プログラミングでは、上位のモジュールから下位の詳細な実装のモジュールを呼ぶということがしばしばあります。しかし、この場合、上位のモジュールが下位のモジュールに依存するこ

とになります。つまり、下位の実装の変更が上位のモジュールにまで影響することになってしまうわけです。

依存関係逆転の原則というのは、上位のモジュールの側に下位のモジュールが実装すべき抽象を定義し、下位のモジュールはその抽象に対して実装を提供すべきという考え方です。たとえば `UserService` が上位のモジュールだとすると、`UserRepositoryImpl` は下位のモジュールになります。依存関係逆転をしない場合、`UserService` から直接 `UserRepositoryImpl` を呼ばざるをえません。このままだと先ほど述べたような問題が生じてしまうので、依存関係逆転の原則に従って、上位のモジュールに抽象的な `UserRepository` を用意し、`UserService` は `UserRepository` を使うようにします。そして、依存性の注入によって、下位のモジュールの `UserRepositoryImpl` を `UserService` に渡すことにより、このような問題を解決できるわけです。

また見落されがちな点ですが、DI では**クライアントに特別な実装を要求しない**という点も重要です。これは Java 界隈で DI が登場した背景に関連するのですが、Spring Framework などの DI を実現する DI コンテナは複雑な Enterprise JavaBeans(EJB) に対するアンチテーゼとして誕生しました。複雑な EJB に対し、何も特別でないただの Java オブジェクトである Plain Old Java Object(POJO) という概念が提唱され、わかりやすさや、言語そのものの機能による自由な記述が重視されました。DI の登場にはそのような背景があり、クライアントに対して純粋な言語機能以外求められないことが一般的です。

最後に、先ほども触れましたが**依存オブジェクトのモック化によるユニットテストが可能になる**という点です。たとえば Web アプリケーションについて考えると、Web アプリケーションは様々な外部システムを使います。Web アプリケーションは MySQL や Redis などのストレージを使い、Twitter や Facebook などの外部サービスにアクセスすることもあるでしょう。また刻一刻と変化する時間や天候などの情報を使うかもしれません。このような外部システムが関係するモジュールはユニットテストすることが困難です。DI を使えば外部システムの実装を分離できるので、モックに置き換えて、楽にテストできるようになります。

以上、DI の利点を見てきました。実装オブジェクト（Dependency）を取得し、サービスに渡す（Injection）という役割をするだけのインジェクタの登場により様々なメリットが生まれることが理解できたと思います。DI は特に大規模システムの構築に欠かせない技術であると言っても過言ではないと思います。

トレイトの自分型を使った依存性の注入の実現

依存性の注入を実現するには様々な方法があります。単純に依存オブジェクトをコンストラクタやセッターメソッドを使ってクライアントに渡す方法や、Java でよく用いられる Spring や Guice などの DI コンテナを使う方法もあります。今回は Scala らしい手法の 1 つであるトレイトの自分型を使った依存性の注入の方法を紹介します。

自分型は、トレイトが自分自身が持つ型を指定できるというものでした。そこで今回は `UserService` に、`UserService` が利用する `PasswordService` と `UserRepository` の型を持たせるようにしま

しょう。

```
trait UserService {
  self: PasswordService with UserRepository =>

  val maxNameLength = 32

  def register(name: String, rawPassword: String): User

  def login(name: String, rawPassword: String): User
}
```

これで UserService の中で PasswordService と UserRepository の機能を利用できるようになりました。ということは UserServiceImpl の実装ロジックもこちらに移すことが可能になったので移してしまいましょう。

```
trait UserService {
  self: PasswordService with UserRepository =>

  val maxNameLength = 32

  def register(name: String, rawPassword: String): User = {
    if (name.length > maxNameLength) {
      throw new Exception("Too long name!")
    }
    if (find(name).isDefined) {
      throw new Exception("Already registered!")
    }
    insert(User(name, hashPassword(rawPassword)))
  }

  def login(name: String, rawPassword: String): User = {
    find(name) match {
      case None       => throw new Exception("User not found!")
      case Some(user) =>
        if (!checkPassword(rawPassword, user.hashedPassword)) {
          throw new Exception("Invalid password!")
        }
        user
    }
  }
}
```

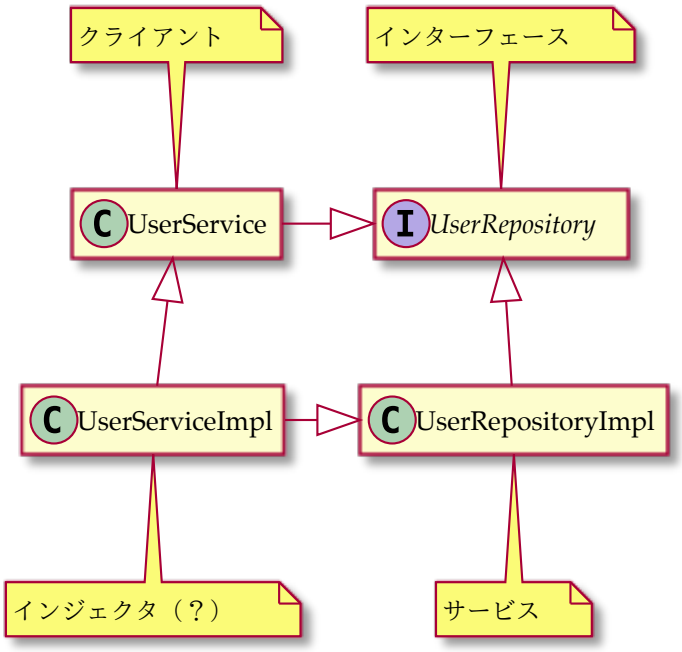
ここで注目すべきなのは、この UserService は PasswordServiceImpl や UserRepositoryImpl への参照を持たないということです。

そして、最後にトレイトのミックスインを使って実装を組み合わせます。

```
class UserServiceImpl extends UserService with PasswordServiceImpl with UserRepositoryImpl
```

これにより依存性の注入が実現できました。

この状態で先ほどのクラスについてクラス図を見てみます。



`UserServiceImpl` の実装が `UserService` に移動したので、クライアントの役割が `UserService` に変わりましたが、`UserServiceImpl` がトレイトのミックスインによりインジェクタのような役割を果たすことで依存性の注入が実現できていることがわかります。

リファクタリング後のユニットテスト

最後に、依存性の注入によりリファクタリングされたことでユニットテストが可能になったので、テストを作成してみましょう。

ここでは `register` メソッドの 2 つのチェックが有効に機能しているかどうかのテストを作成します。

テストには引き続き、ScalaTest を使います。

include

`sut` はテスト対象となる `UserService` のインスタンスです。`PasswordServiceImpl` は外部のシステムを使わないので、そのまま使っています。しかし、`UserRepositoryImpl` は実際のデータベースシステムを必要とするのでユニットテストでは使うことができません。なので、ここでは仮の実装として、すべてのメソッドで `user` というテスト用のユーザーオブジェクトを返すようにします。このようにトレイトを使うことで、テスト用の実装に入れ替えることが可能になったわけです。

2 つのテストではそれぞれ、長すぎる名前を与えられたときにちゃんとチェックされ例外が発生し

ているかどうか、名前が既に登録された場合にちゃんとチェックされ例外が発生しているかどうかがテストされています。