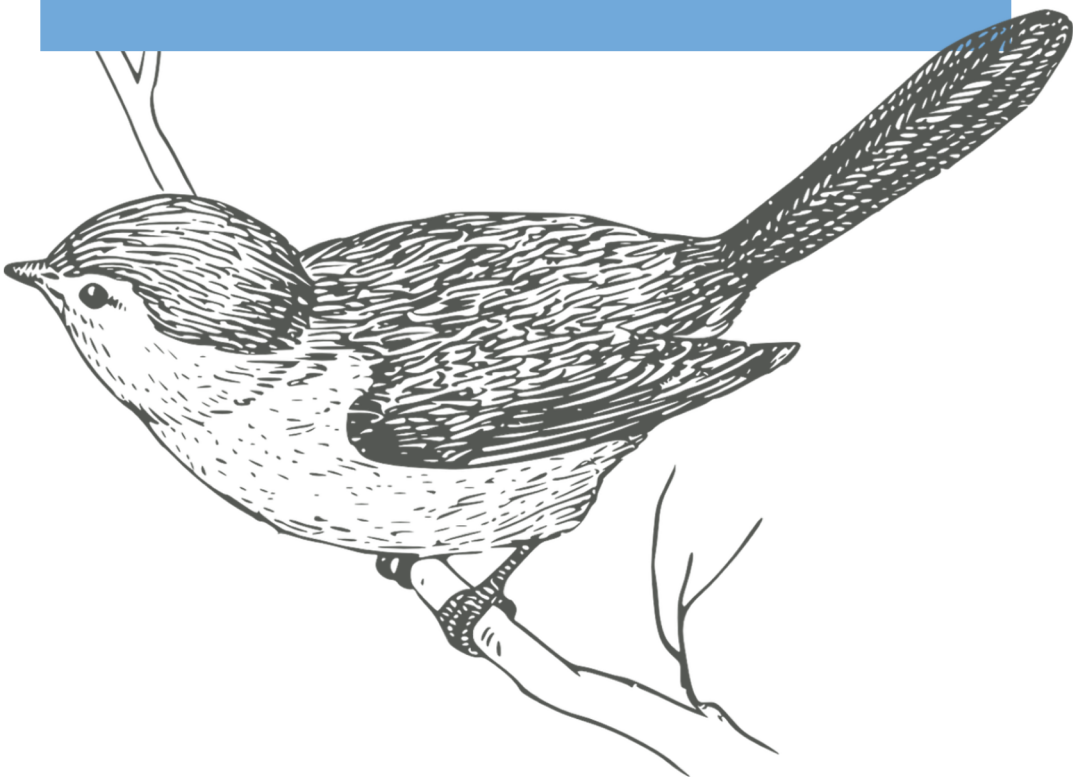
日本Kotlinユーザグループ

入門までの

Kotlin 助走読本



長澤太郎 他9名 著

はじめに

思わず絶叫していました。2017 年 5 月 18 日未明（JST）、私が友人たちと一緒に Google I/O の Keynote の生中継を観ていたら、思いがけない朗報が発表されました。Android アプリ開発言語として「Kotlin」が正式サポートされるというニュースです！ 私は以前からの Kotlin の大ファンで、この瞬間を何年も待ち望んでいました。この Google の決定は、世界中の多くの Kotlin ユーザーに勇気を与えてくれるものです。今こそ、Kotlin を始める時が来たのです！

とは言っても、今回の発表ではじめて Kotlin というプログラミング言語を耳にした開発者も多くいることでしょう。「Kotlin とは何だ？」とか「なんで Kotlin なのか？」とか「どうやって学べばいいのか？」という戸惑いもあるかと思います。そこで Kotlin への情熱に突き動かされた愛の戦士（エンジニア）たちが集い、「多くの人に Kotlin の良さを知ってもらいたい」という想いで本書を執筆しました。

本書はタイトルのとおり「入門までの助走」をサポートします。サクサク読めるようボリュームを少なく抑え、サンプルコードをごくシンプルなものに留めています。3 章で構成し、読み物として楽しめる導入部分から始まります。コードを交えた具体的な解説に続き、最後に次のステップを示して締めくくります。

書籍の体裁を取ったのは、筆者の自信の現れでもあります。Kotlin は楽しいプログラミングライフをもたらしてくれるでしょう。本書がその手助けになれば、我々にとってこれ以上の幸せはありません。

2017 年 5 月吉日 著者代表 長澤 太郎

目次

第 1 章

Kotlin について知る

本章は「Kotlin とはどういうプログラミング言語なのか」という疑問に対する回答です。特徴や誕生の背景、これからどうなっていくのか、実際に使用している企業を紹介します。あまりコードは登場しませんが、雰囲気を知るには打ってつけの内容です。

1.1 Kotlin とは

Kotlin（ことりん）[*1]は、JetBrain 社が Java に代わり、より簡潔に書くことを目的として作られた **JVM**（Java 仮想マシン）上で動作する静的型付き言語で、JavaScript へのトランスパイル[*2]も可能です。また、2017 年 5 月現在はまだプレビュー版ですが、LLVM toolchain を使ってネイティブコードにコンパイルすることで iOS 等のクロスプラットフォームでの動作を目指しています [1]。構文は Java と互換性がありませんが、Kotlin は Java コードと相互運用するように設計されており、コレクションフレームワーク等の既存の Java クラスライブラリーの Java コードに依存しています。既存の Java プロジェクトでも一部のファイルのみを Kotlin で書くことができるので、移行のハードルが低いのも特徴で

[*1] Kotlin という名前は JetBrains 社が拠点とするロシアのサンクトペテルブルク近郊のコトリン島に由来します [2]

[*2] ソースコードからソースコードへ変換すること

しょう。さらに、Kotlin を開発している JetBrains 社は IntelliJ IDEA と呼ばれる **IDE**（統合開発環境）を開発していることも大きな強みです。IDE のサポートがあることで、言語の実装とツールのサポートの両方で恩恵を受けられます。

思想

Kotlin は他の主要言語と比較して歴史が浅く、さまざまな言語の影響を受けています。Kotlin のメイン開発者である Andrey Breslav は Java、Scala、C #、Groovy に大きな影響を受け、他にも Objective-C、Gosu、Spec #、Python、Eiffel 等の言語を参考にしたと述べています [3]。他の言語の良い部分を積極的に取り入れることで、モダンで書きやすい言語を目指しているように思えます。Kotlin の目標のひとつとして、Java と同等のコンパイル速度にすることを掲げています [4]。現状はまだ Java と同等のコンパイル速度ではありませんが、意識するほど遅くはないため実務上使用しても何ら問題のないレベルのスピードです [5]。また、Android での利用も強く意識しており、Kotlin を導入する障壁となる 64K メソッド数問題[*3]を解決するため、ランタイムのメソッド数が少ないのも大きな特徴です。

歴史

Kotlin は 2010 年頃に開発を開始して、2012 年 2 月 14 日に Apache license Version 2.0 にてオープンソース化されました。その後 2012 年 4 月 12 日にマイルストーンの Version 1 として Kotlin M1 をリリースしています。当初は Java の代替言語として開発されていた Kotlin ですが、2013 年 8 月にリリースした M6 からは Android Studio をサポートし、M11 で Android Extensions 等の機能が追加され Android 開発を強く意識した言語になりました。2015 年 10 月の M14 リリースのあと 1.0-beta1~4、1.0-RC を経て 2016 年 2 月 15 日 1.0 がリリースされました。2017 年 5 月現在の最新バージョンは 1.1.2 になっています。1.0 リリースと同時に主にサーバーサイドで多く利用されている Spring Boot の

[*3] Android では 65536 (64 × 1024) のメソッドカウント制限があります https://developer.android.com/studio/build/multidex.html

正式サポートも発表しています。1.0 の正式リリースにより、国内外の企業で主に Android アプリ開発への利用実績が認められ、ついに「Google I/O 2017」という世界的なカンファレンスで、Google が Android での Kotlin 公式サポートを発表しました。M1 から Kotlin 1.1.0 までの Kotlin の変化の過程を表 1.1 に示します。

1.2 Kotlin を使うと何が嬉しいのか

Kotlin がリリースされて以来、Google の公式サポートがないにもかかわらず、一部の Android 開発者の中で熱狂的に支持されてきました。彼ら（例外もいますが）は、新しいものにただ飛びついたわけではありません。Kotlin は次の点で優れており、採用が進んでいます。

- Android 開発との親和性の高さ
- 生産性の高さ
- メンテナビリティの高さ

具体的な例は後述するとして、ここではそれぞれの項目について概要を説明します。

Android 開発との親和性の高さ

Kotlin と Android の親和性は非常に高いです。Kotlin は特に何も設定せずに、直接 Java のコードを呼び出すことができます。また Java からも Kotlin のコードを呼び出すことが可能です。すでに多くのプロジェクトで人気がある Retrofit[*4] のような Java 製ライブラリーを利用することも可能です。Kotlin ですべてを書き直す必要はなく、既存の Java プロジェクトに Kotlin を途中から導入することもできます。JVM 上で動く他の言語でも同じことが可能なのですが、Kotlin はそれらに比べ次の点で Android 開発との相性の良さが伺えます。

- メソッド数が少ない

[*4] タイプセーフな HTTP クライアントライブラリ http://square.github.io/retrofit/

- JetBrains 社による強力な IDE のサポートがある
- ライブラリのサイズが小さい（Kotlin v1.1.2-2 のライブラリサイズは 859KB）

生産性の高さ

Kotlin の生産性は次の点で明らかに向上します。

- 少ない記述量
- コレクション操作などの便利な関数
- Java にはない有用な機能

Kotlin は後発の言語ということもあり、文法が洗練されています。極力、冗長なコードを書くことなくアプリ開発を行うことが可能です。たとえば `id`, `name` を持った `Person` クラスの定義を Java で書く場合、`getId()`, `getName()`, `setName(String name)`, `toString()`, `hashCode()`, `equals(Object)` メソッドを用意する必要がありました。Kotlin では `data` クラスを使うことにより、一行書くだけでそのすべての実装を Kotlin が生成します。

シングルトンを使ったことがありますか？ Kotlin では `class` を `object` と書き換えるだけでシングルトンの実装が完了します。

他にもいろんな工夫が散りばめられています。これらの工夫のおかげで、Java と比較し多くのプロジェクトでコード量を削減することが可能です。

Java ではコレクション操作が大変ですが Kotlin ではコレクション操作のための多くのメソッドが用意されています。これらのメソッドのおかげでシンプルかつ直感的に書けるようになります。Kotlin でコレクションを操作する場合、`for` ループを使うことは滅多にありません。Java での Android アプリ開発時は Retrolambda[*5]を導入しない限り利用することができなかった、より短く記述できるラムダ式にもデフォルトで対応しています。

Java にはない便利な文法も多数存在します。拡張関数、遅延初期化、演算子

[*5] Java7 で Lambda を使うためのライブラリ https://github.com/orfjackal/retrolambda

オーバーロード、エルビス演算子などなど盛りだくさんです。一度これらを使ってしまえば、なぜ Java にこれらが存在していないのかと疑問に思うはずです。

メンテナビリティの高さ

Kotlin は変数定義時に、この変数が「読み込み専用か上書きできるかどうか？」と「Null 許容型か Null 非許容型かどうか？」を定義する必要があります。読み込み専用変数を定義し、その変数を上書きするようなコードを書いてみて下さい。Kotlin ではこのようなコードはコンパイルエラーになります。同様に Null 非許容型の変数に `null` を代入してもコンパイルエラーとなります。逆に上書きできる変数定義をして、実際にその変数が上書きをしていなかった場合、読み込み専用変数へと定義を変更するように IDE が警告してくれます。変数の再代入を極力避けるようなコードを、チーム全体が特に苦なく書けるようになります。Kotlin の変数定義だけを見ても `NullPointerException` や再代入を避けたメンテナビリティの高いコードが書けるように、言語レベルで設計されています。

1.3 Kotlin の特徴

静的型付けのオブジェクト指向言語

Kotlin は Java と同じく静的型付けの言語です。静的型付けとはコンパイル時など、プログラムの実行前に型が決まることです。実行するまで、ある変数に文字列が入っているのか、数値が入っているのか分からないということはありません。そして `NullPointerException` をはじめとする、`null` にまつわる問題をうまく扱うことができる言語機能も提供されています。

また、Java と同じくオブジェクト指向言語です。オブジェクト指向とはオブジェクトとメッセージの相互作用をもとにした考え方です。この考え方を踏襲した言語のことをオブジェクト指向言語と言います。

Kotlin がターゲットとするプラットフォーム

Kotlin はフルスタックな言語を目指し開発が行われています。Android を始めとする JVM 上で使用可能です。もちろんサーバーでも使用可能です。

一方で、JVM 上以外でも動作するようにも開発されています。その内の 1 つが、バージョン 1.1 でサポートされた JavaScript へのトランスパイルです。Web フロントエンド開発は多くのエコシステムのもとに成り立っており、現在の Web フロントエンド開発で用いられている **npm** や **webpack**、**React** 等の使用も可能となっています。これらにより、Kotlin を用いてサーバー・Android・JavaScript までフルスタックに開発することが可能になりました。

そして、現在は Kotlin を LLVM コンパイラーを通して機械語にコンパイルできる「Kotlin/Native[*6]」が開発されています。開発者はソースコードを記述し、特定のプラットフォームで動作可能な機械語に変換する必要があります。このとき、プログラミング言語によってコンパイル方法が異なります。LLVM はその依存を吸収するために、プログラミング言語とプラットフォームの両方から独立する中間コードを生成します。プレビュー版では次のプラットフォームがターゲットとされています。

- Mac OS X 10.10 and later (x86-64)
- x86-64 Ubuntu Linux (14.04, 16.04 and later)
- Apple iOS (arm64)
- Raspberry Pi

LLVM を利用しているので、今後さらなるプラットフォームへの拡張が期待できるでしょう。

[*6] 区別のために **Kotlin/JVM**、**Kotlin/JS**、**Kotlin/Native** と記述することがあります

高階関数

Kotlin では高階関数がサポートされています。高階関数は関数オブジェクトを引数にしたり、戻り値にしたりできます。リスト 1.1 に例を示します。

リスト 1.1: 高階関数の例

```
// body という名前の引数に関数オブジェクトを渡している
fun <T> lock(lock: Lock, body: () -> T): T {
  lock.lock()
  try {
    // 引数で渡された関数オブジェクトを実行している
    return body()
  }
  finally {
    lock.unlock()
  }
}
```

ラムダ式

ラムダ式を使うと関数を宣言せずに関数オブジェクトをすぐに生成できます。リスト 1.2 のコードは高階関数の例ですが、第二引数に注目してください。第二引数にはラムダ式で a、b の引数をとり、比較結果を返すという関数オブジェクトが記述されています。

リスト 1.2: ラムダ式の例

```
  max(strings, { a, b -> a.length < b.length })
```

1.4 将来の展望

2017 年 5 月現在、Kotlin の国内外においての利用は Android アプリが主に多く、次にサーバーサイドの開発でも広く利用されています（実例については、「導入事例」の節をご参照下さい)。そういった現状から Kotlin はどのように発展していくのでしょうか。

JetBrains として将来の展望

Kotlin を開発する JetBrains 社は、今後もモダンな言語機能を提供しつつマルチプラットフォーム開発を実現させることを目標としています。

具体的には Android、iOS、サーバーサイド、Web フロントエンド、ネイティブなど各種プラットフォーム開発を目指しています。

Android と Kotlin

Google I/O 2017 で、Kotlin を Android の開発言語として公式にサポートすることが発表されました。さっそく、**developer.android.com** には Kotlin のページが登場しています[*7]。

Google と JetBrains は協力し、Kotlin を開発するための非営利団体を設立することも同時に発表され、Android における Kotlin サポートの本気度が伺えます。

Kotlin を公式にサポートした理由として、同カンファレンスのキーノートにおいて次のように挙げています。

- Kotlin は Java との相互運用が可能である
- IDE 開発元と同じチームが開発しており、素晴らしい IDE サポートを受けることが可能である
- 言語が成熟し製品版としてリリース可能であり、すでに多くの導入事例がある

[*7] https://developer.android.com/kotlin/index.html

「Kotlin が Android で使えなくなるかもしれない」といった将来的な不安要素が無くなった今、ますます Android での Kotlin 利用は増えていくでしょう（Google としては Java と C++ もこれまでと変わらずサポートし、Kotlin は追加でのサポートとしています）。

推測の域ではありますが、今後どうなっていくかを考えてみましょう。公式言語となったことで、Android 開発者にとっては IDE レベルでのさらなるサポートやツールの改善などが増え、開発の強力な味方となってくれると予想しています。Kotlin を使いこなせるプログラマーの需要が高まることもあるかもしれません。

1.5 導入事例

海外における導入事例

Google I/O 2017 の開発者向けキーノートにおいて Kotlin の製品版導入事例として、次の 4 つの Android アプリが紹介されました。

- Flipboard[*8]
- Pinterest[*9]
- Square Cash[*10]
- Expedia[*11]

また、Kotlin 公式サイトによると、Evernote の Android クライアントは Kotlin への移行をすすめており[*12]、Trello の Android アプリにおいても 2016 年の 12 月より製品版に Kotlin を導入[*13]しています。

[*8] https://play.google.com/store/apps/details?id=flipboard.app
[*9] https://play.google.com/store/apps/details?id=com.pinterest
[*10] Square Cash は日本国内では利用できません
[*11] https://play.google.com/store/apps/details?id=com.expedia.bookings
[*12] https://blog.evernote.com/tech/2017/01/26/android-state-library/
[*13] https://twitter.com/danlew42/status/809065097339564032

日本国内における導入事例

株式会社サイバーエージェントが提供する「FRESH![*14]」というサービスでは、Android クライアント[*15]とサーバーサイドにおいて Kotlin を導入しています。特に Android クライアントは 2015 年の 3 月の開発当初から本書の筆者である藤原と荒谷によって Kotlin で開発され、製品版としてリリースしています。また、同社が提供する「famchatty[*16]」の Android クライアントも Kotlin によって開発され、同社グループ企業である AWA 株式会社が提供する「AWA[*17]」の Android クライアントは Kotlin への移行を進めています。

本書執筆陣による Kotlin での開発事例は他にもあり、山本の所属する Sansan 株式会社が提供する「Eight[*18]」、八木が開発部長を務める株式会社トクバイが提供する「トクバイ[*19]」、長澤の所属するエムスリー株式会社の「MR 君」はいずれも Kotlin で開発されています。株式会社 i-plug の提供する「OfferBox[*20]」の Android アプリは本書執筆陣のきの子によって Kotlin で 2017 年 5 月にリニューアルされました。

また、次のような会社で Kotlin は導入されています。

- Retty 株式会社「Retty[*21]」
- 株式会社エウレカ「Pairs[*22]」
- 株式会社コンセプト[*23]
- 株式会社 FOLIO[*24]（導入予定）

[*14] https://freshlive.tv/
[*15] https://play.google.com/store/apps/details?id=jp.co.cyberagent.valencia
[*16] https://play.google.com/store/apps/details?id=jp.co.cyberagent.malaga
[*17] https://play.google.com/store/apps/details?id=fm.awa.liverpool
[*18] https://play.google.com/store/apps/details?id=net.eightcard
[*19] https://play.google.com/store/apps/details?id=jp.co.tokubai.android.bargain
[*20] https://play.google.com/store/apps/details?id=jp.offerbox
[*21] http://engineer.retty.me/entry/retty-to-kotlin
[*22] https://developers.eure.jp/tech/pairs-meets-kotlin/
[*23] http://qiita.com/omochimetaru/items/98e015b0b694dd97f323
[*24] https://folio-sec.com/

- フラー株式会社「Joren [*25]」

[*25] https://joren.io/ja/

表 1.1 Release history

Version	Release	Note
M1	2012/04/12	IntelliJ の公式プラグインとして Kotlin が登場した
M2	2012/06/11	`!!` assert not null, `invoke()`
M3	2012/09/20	data class, Enum, Annotation の改善, sure の削除
M4	2012/12/11	Tuple の削除, data class のコピー, 型推論やコード補完等の IDE 面での大幅な改善
M5	2013/02/04	デフォルト name space の変更, 内部クラスは static に, Float リテラル
M6	2013/08/12	Android Studio サポート, SAM 変換, Annotation が Enum と可変長引数対応
M7	2014/03/20	inline 関数
M8	2014/07/02	Standard Library の強化 (Slice, Join, etc), JS でも Data class が使用可能に
M9	2014/10/15	Incremental build, platform type, local object の削除
M10	2014/12/17	Reified type parameters, JS の改善 (Dynamic, Annotation)
M11	2015/03/19	複数コンストラクター, init, companion object, kotlin-android-extension
M12	2015/05/29	kapt1, trait->interface, [interface 継承, class object, break, continue] の削除
M13	2015/09/16	lateinit, seald, デフォルトスコープが public に, Java の getter, setter サポート
M14	2015/10/01	operator, const
1.0-beta1	2015/10/22	infix, Java との互換性意識
1.0-beta2	2015/11/16	Collection library, IntRange の改善等
1.0-beta3	2015/12/07	言語の bugfix: when 文での or 文変更, Enum の values() 削除, android kotlin plugin の統合
1.0-beta3	2015/12/22	Incremental compile 改善, overload の解決アルゴリズム改善, `(R) -> T` で R のプロパティ参照
1.0-RC	2016/02/04	リリースに向けての最終調整
1.0	2016/02/15	正式リリース, Spring Boot のサポート
省略		
1.0.4	2016/09/22	kapt2
1.0.6	2016/12/27	kapt3
省略		
1.1.0	2017/03/01	coroutine, typealias

第 2 章

Kotlin を学ぶ

本章では、Kotlin の文法や機能をコードを交えながら具体的に解説します。とは言っても、詳細には立ち入りません。眺めるだけでなんとなく理解できるように努めました。

2.1 環境構築から HelloWorld まで

この節では、実際に Kotlin を動作させて試してみます。

Kotlin を試してみるにはいくつかの方法がありますが、本書では次の 2 つを紹介します。

- Web 上の実行環境である try.kotlinlang.org で試してみる方法
- Android Studio 上で Kotlin 環境を構築して動作させてみる方法

try.kotlinlang.org を使う

Kotlin を試してみる一番手っ取り早い方法は、ブラウザー上で `https://try.kotlinlang.org` にアクセスしてみることです（図 2.1）。

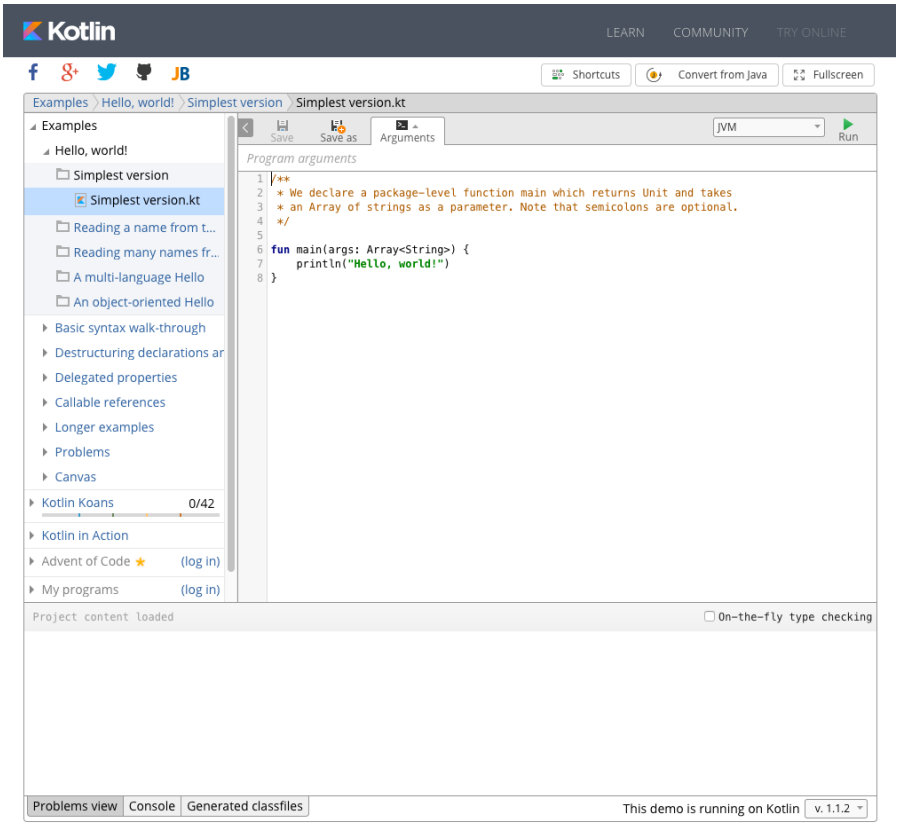


図 2.1 https://try.kotlinlang.org

try.kotlinlang.org は Web 上の Kotlin 実行環境です。このページにアクセスするとまず"Hello, world!"のサンプルプログラムが表示されます（リスト 2.1）。

リスト 2.1: Hello, world! by Kotlin

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

ページの右上の「▶ Run」と書かれたボタンを押してみましょう。すると、このプログラムが実行され画面下部のコンソール部分に

```
Hello, world!
```

と出力されます。

Kotlin の Hello world!

このたった 3 行の"Hello, world!"プログラムですが、Kotlin は Java に比べてシンプルにプログラムを記述できることが分かるでしょう。比較のためリスト 2.2 に、Java の HelloWorld プログラムを載せてみます。

リスト 2.2: Hello world! by Java

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

■トップレベルの関数 Java の場合、エントリーポイントとなる `main` メソッドを定義するためには、まず何かしらのクラスを定義して、その `static` メソッドとして `main` メソッドを定義する必要がありました。一方 Kotlin の場合は、トップレベルに関数を定義することができます。

■関数定義は fun で始まる すでにお気づきだと思いますが、Kotlin の `main` 関数は `fun` というキーワードで始まっています。これはトップレベルの関数でも、クラスのメソッドでも同じ決まりです。

■引数の型は後ろに `main` 関数の引数は `args: Array<String>`のように定義されています。これは `args` という名前で、引数の型が `Array<String>`（`String` の配列）であることを示しています。このように Kotlin では、引数や変数を定義する場合にはまずその名前を定義し、その後に:（コロン）に続けて型を定義する方式をとります。

■戻り値の型を省略可能 `main` 関数には戻り値の型の指定がありません。Kotlin では、意味のある戻り値がない関数やメソッド（Java でいう `void` 型のメソッドに相当）の戻り値の型は `Unit` 型と定義され、その記述は省略することができます。

この関数に戻り値の型を明示的に指定する場合には、リスト 2.3 のような指定になります。

リスト 2.3: 戻り値の型の指定がある場合

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

関数やメソッドの戻り値の型も引数や変数と同じように、関数宣言の後ろに:に続けて定義するようにします。ここでは `main` 関数の戻り値の指定は `Unit` 型となり、この関数の中では `Unit` 型のシングルトンオブジェクトを返しています。

■セミコロンなし `println("Hello, world!")` の文末には;（セミコロン）がありません。Kotlin では文末の; は不要です。

Kotlin のコードを書いてみる

では、この try.kotlinlang.org を使って、もう少し Kotlin の機能ををつかったコードを書いてみましょう。もとの `Hello, world!`のコードをリスト 2.4 のように変更してみます。

リスト 2.4: args を出力する

```
fun main(args: Array<String>) {
    println("Hello, ${args[0]}!")
}
```

ここで「▶ Run」ボタンを押して実行すると

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
    at Simplest_versionKt.main(Simplest version.kt:7)
```

とエラーが出力されてしまいます。このコードを正しく実行するためには図 2.2 のように、**Program arguments** の欄に何かしらの文字列（ここでは"Kotlin"という文字列）を指定します。

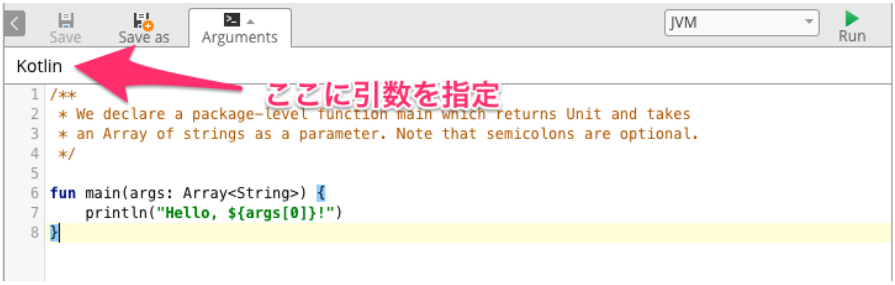


図 2.2　プログラム引数を指定

そして「▶ Run」ボタンを押して実行すると

```
Hello, Kotlin!
```

と **Program arguments** で指定した文字列が出力されました。

■String テンプレート　リスト 2.4 で使った機能は、Kotlin の「String テンプレート」という機能です。文字列中に$（ドル記号）とともに変数や式を挿入すると、それを評価した値を文字列と連結してくれます。リスト 2.5 は単独の変数 i を使用した例です。

リスト 2.5: 単独の変数を使った例

```
val i = 42 // 変数 i に 42 を代入
val s = "すべての答え:$i" // -> すべての答え:42
```

リスト 2.5 の val i = 42 の文では変数の宣言と代入を行っています。val で宣言された変数は、それが再代入が不可（読み取り専用）な変数であることを表します。変数 i には 42 という値を代入していますが、42 は Int 型のリテラルであるため、変数 i の型も Int 型である、と推論されています。そして次の行の変数 s には変数 i の値が含まれた文字列が代入されます。変数 s の型は String 型と推論されます。

さらにリスト 2.6 のように、$の後ろに{}（波括弧）をつけ、そこに何かしらの値を出力する式や、オブジェクトのプロパティなどを記述することでシンプルな変数だけでなく、もう少し複雑な式も文字列に含めることができます。

リスト 2.6: 複雑な式を使った例

```
val s1 = "1 + 1 は${ 1 + 1 }です。"      // 1 + 1 は 2 です。
val s2 = "s1 の長さは${s1.length}です。" // s1 の長さは 10 です。
```

リスト 2.4 のコードも、この機能をつかって記述しています。

■コラム: タイトル

この try.kotlinlang.org では Hello, world!のサンプルプログラムの他に、

次のコード群が含まれています。

- Kotlin の文法サンプル
- 「Kotlin Koans」というチュートリアル（第 3 章 1 節参照）
- 「Kotlin in Action」という Kotlin 入門書（英語）のサンプルプログラム

Kotlin の学習にとても役立つものになっています。ぜひ試してみてください。

Android Studio 上に Kotlin 環境を構築して動作させる

次に、すでにインストールされている Android Stdio に対して Kotlin の環境を構築し、実際に Kotlin で Android のアプリを実装して動作させるまでのコードを説明します。大まかの手順は次のとおりです。

1. Android Studio に Kotlin プラグインをインストール
2. 新規 Android アプリのプロジェクトを作成
3. Java のソースコードを Kotlin に変換

ここでは

- Android Studio 2.3.2

を対象に環境の構築していきたいと思います。

プラグインのインストール

Android Studio で Kotlin を使用するには、Kotlin プラグインをインストールする必要があります[*1]。プラグインのインストールは次の手順で行います。

- プラグインをインストールするには Android Studio のメニューから [Android Studio] -> [Preference] を選択
- Android Studio 設定ダイアログを表示（図 2.3）
- 設定メニューから [Plugins] を選択し、Plugin 管理設定を開く

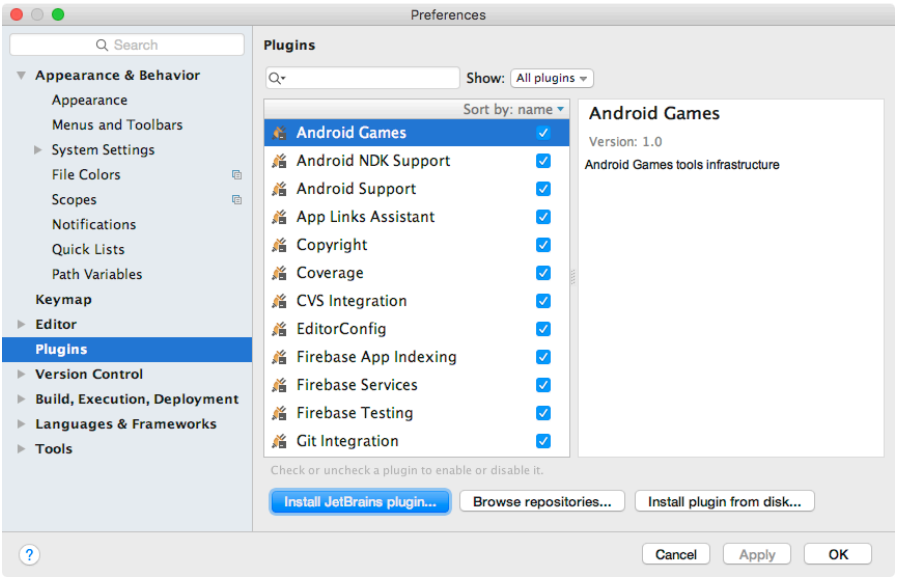


図 2.3　Android Studio 設定ダイアログ

- Plugin 管理設定にて [Install JetBrains plugin…] ボタンをクリックし Browse JetBrains Plugins 選択ダイアログ（図 2.4）を開く

[*1] Android Studio の 3.0 以降のバージョンでは Kotlin が標準でバンドルされているため、この手順は必要ありません

- Browse JetBrains Plugins 選択ダイアログの検索窓から `Kotlin` と入力し Kotlin プラグインを選択
- Kotlin プラグインの Install ボタンをクリックし Kotlin プラグインをインストール

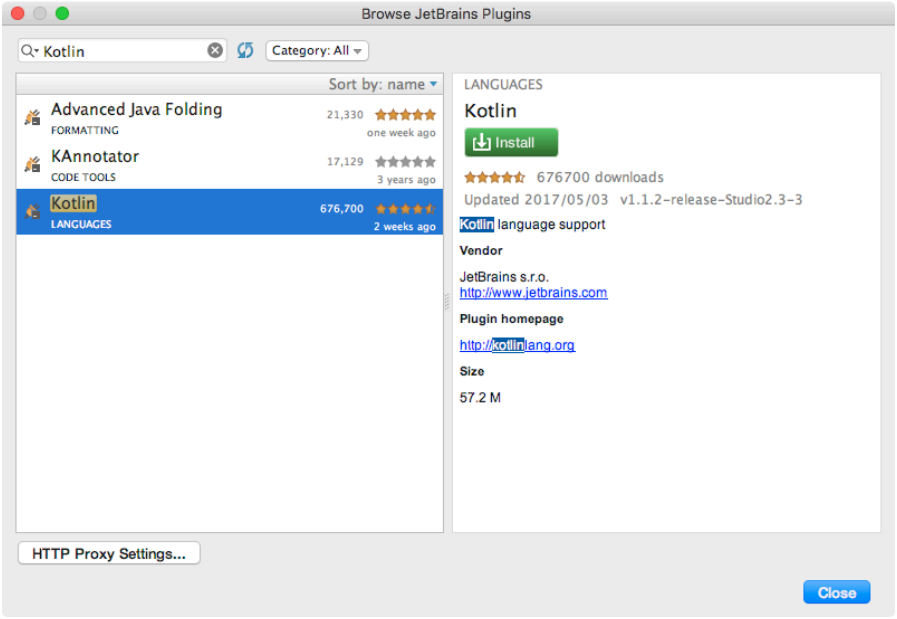


図 2.4 Browse JetBrains Plugins 選択ダイアログ

- インストールが完了したら [Restart Android Studio] ボタンをクリックし Android Studio を再起動

お疲れ様でした。これで Android Studio で Kotlin を書く準備ができました。

サンプルアプリケーションの作成

次に土台となる Android アプリケーションを作成します。

Android Studio のメニューから [File] -> [New] -> [New Project...] を選択し

て、新規プロジェクト作成ウィザードを開始します。

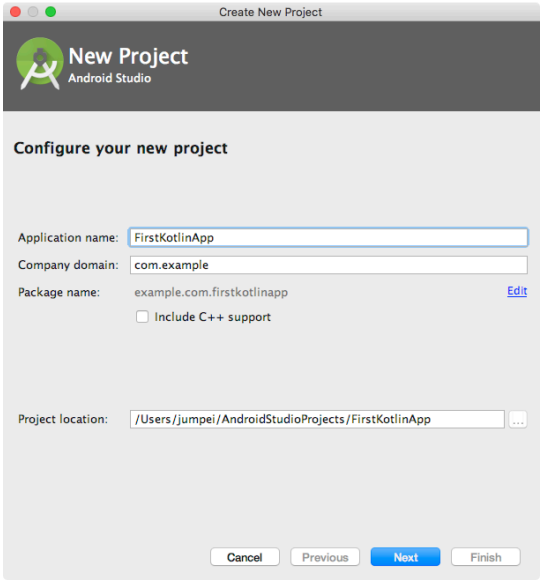


図 2.5　アプリーケーション名の入力

適当なアプリケーション名を指定したら Target Device と Minimum SDK バージョンを指定します。ここでは Target Device に [Phone and Tablet] を指定し、Minimum SDK バージョンは API 15 を指定しています。Kotlin を使用するにあたって、Minimum SDK バージョンは特に制約はありません。最小バージョンである API 9 から使用できます。

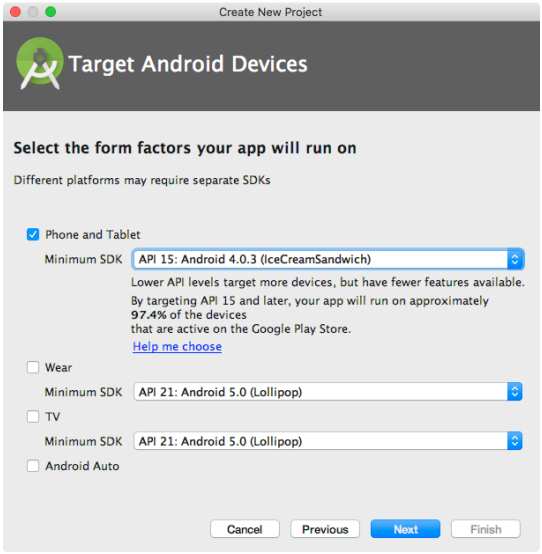


図 2.6　Target Device を SDK バージョンの指定

次に、Activity の指定ではもっともシンプルな [Empty Activity] を選択します。

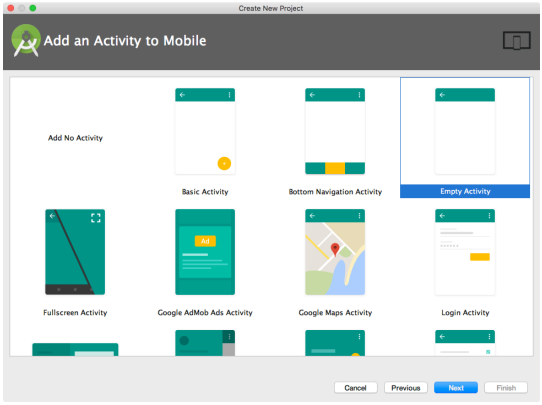


図 2.7　Activity の選択

最後に Activity のクラス名を指定して、新規プロジェクトの作成は完了します。ここは規定値の `MainActivity` のままにしておきます。

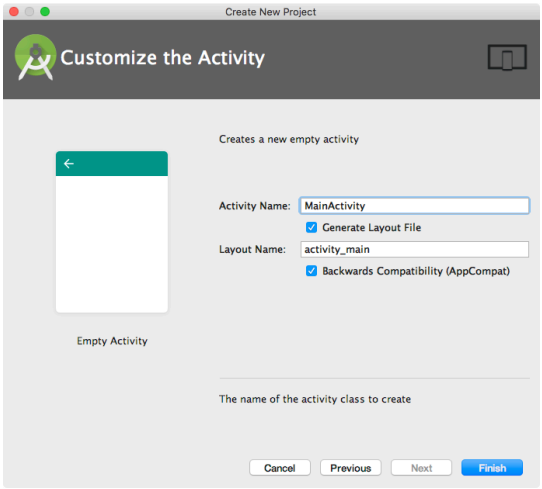


図 2.8　Activity 名の設定

これで土台となる Android アプリケーションが作成されました。

Java ファイルの Kotlin へのコンバート

ここでは、作成した Android のアプリケーションの Java コードを Kotlin に変換していきます。

Android Studio にて `MainActivity` のファイルを開いている状態で、メニューから [Code] -> [Convert Java File to Kotlin File] を選択しファイルを Kotlin に変換します。

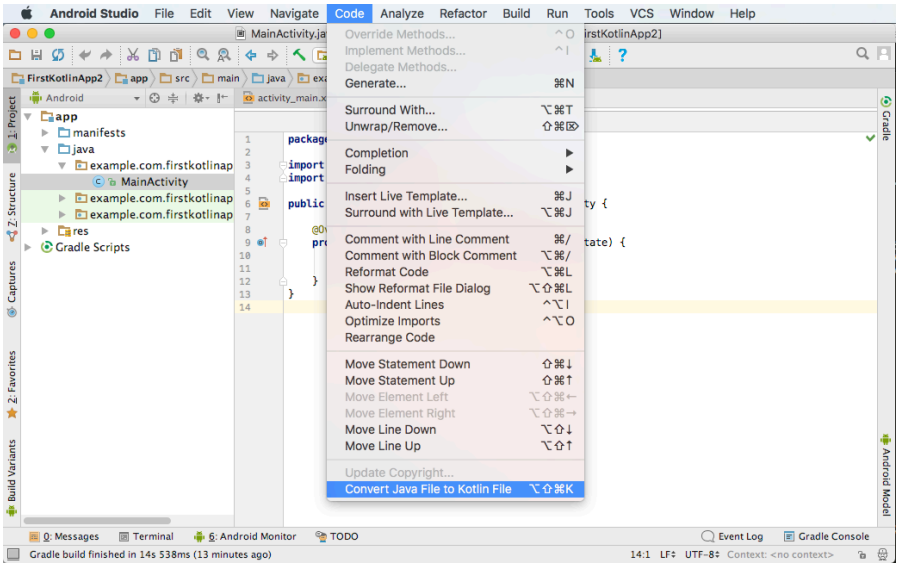


図 2.9 Convert Java File to Kotlin File メニューの選択

これで Java のファイルを Kotlin に変換できました。

しかし、この状態では Android アプリケーションのプロジェクトでの Kotlin の設定ができていないため

Kotlin not configured

のメッセージが表示されますので、このメッセージからプロジェクトに対して Kotlin の設定を行います。まず [Configure] のボタンを選択し、[Choose Configurator] のメニューはそのまま [Android with Gradle] を選択します。

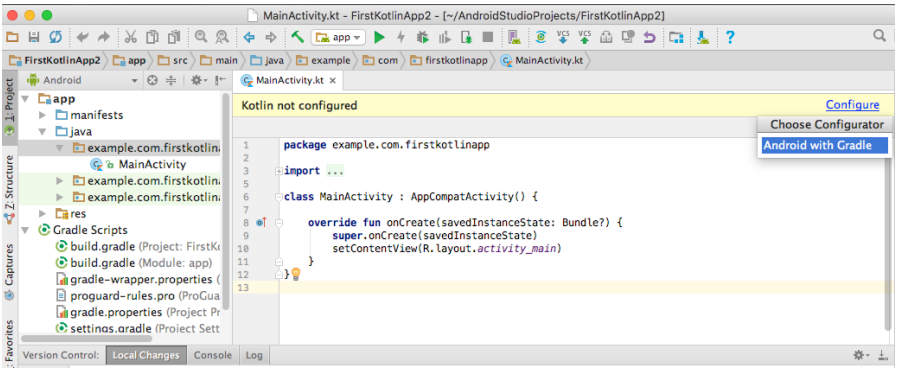


図 2.10　Kotlin not configured のメッセージからの設定

その次の [Configure Kotlin in Project] ダイアログでは

All modules containing Kotlin files: app

を選択し、

Kotlin conpiler and runtime version:

は、先ほどインストールしたプラグインと合わせるため 1.1.2-3 を選択します。

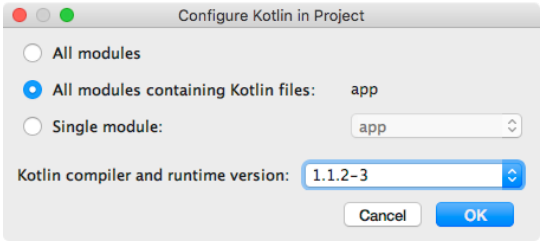


図 2.11　Configure Kotlin in Project ダイアログ

これで Android プロジェクトの設定は完了しました。

Kotlin で書かれた Activity クラス

では、早速 Kotlin に変換されたコードを見ていきましょう。Android Studio によって変換された結果、リスト 2.8 のようなコードができあがっているはずです。比較のために Java で書かれたコード（リスト 2.7）も載せておきます。

リスト 2.7: Java で書かれた MainActivity クラス

```
 1: package example.com.firstkotlinapp;
 2:
 3: import android.support.v7.app.AppCompatActivity;
 4: import android.os.Bundle;
 5:
 6: public class MainActivity extends AppCompatActivity {
 7:
 8:     @Override
 9:     protected void onCreate(Bundle savedInstanceState) {
10:         super.onCreate(savedInstanceState);
11:         setContentView(R.layout.activity_main);
12:     }
13: }
```

リスト 2.8: Kotlin に変換された MainActivity クラス

```
 1: package com.example.firstkotlinapp
 2:
 3: import android.support.v7.app.AppCompatActivity
 4: import android.os.Bundle
 5:
 6: class MainActivity : AppCompatActivity() {
 7:
 8:     override fun onCreate(savedInstanceState: Bundle?) {
 9:         super.onCreate(savedInstanceState)
10:         setContentView(R.layout.activity_main)
11:     }
12: }
```

このコードを見てみると、さらに Kotlin の特徴的な仕様が見えてきます。

■クラス定義 ここでははじめてクラスの定義が出てきます。クラスの定義は Java と同様に `class` のあとにクラス名を宣言します。また、その後ろに:に続けて継承元のクラスを記述します。`class` の前に何も指定しない場合は、`public` スコープなクラスとなります。

■クラスのメソッド 8 行目の `onCreate()` は、このクラスのメソッドです。関数のときと同じように `fun` を使って定義します。Java の場合にオーバーライドするメソッドは、`@Override` というアノテーションをつけていましたが、Kotlin でも `override` という修飾子を `fun` の前に付ける必要があります。

■Null 許容型 また、`onCreate()` メソッドの引数は `Bundle?`型である、と定義されています。この `Bundle` の後ろにある`?`は Null 許容型であることを示します。Null 許容型であるということはこの変数には `null` が入り得るということです。詳細は後述します。

2.2 Kotlin の味見

基本文法

変数の宣言

変数の宣言のキーワードは 2 つあります。`val` と `var` です。次のように宣言します。

リスト 2.9: 変数の宣言

```
val num : Int = 1
var num2 : Int = 2
```

明示的に型を宣言するには、変数の後ろに:と型名をつける必要があります。

Int は Kotlin の数値型です。

val は変数の再代入ができません。var の場合は、変数の再代入が可能です。

リスト 2.10: val への再代入 (コンパイルエラー)

```
val num : Int  = 1
num = 2 // =>コンパイルエラー
```

リスト 2.11: var への再代入

```
var num2 : Int = 2
num2 = 3 // =>コンパイルエラーにならない
```

型の省略

先ほどの val num :Int = 1 の場合、実は型の宣言が省略可能です。型の省略が可能なのは右辺が 1 なので、Int 型であるとコンパイラーが推論しているためです。

リスト 2.12: 型の省略

```
val num = 1
```

型を省略できるのは便利ではありますが、開発者同士で分かりにくい場合は明示的に宣言する方がよいでしょう。

条件判定（if 文）

条件判定には if 文を使います。

リスト 2.13: Java の if 文のような使い方

```
if(true) {
    println("true")
}
```

リスト 2.13 のように、Java などの if 文と同じような使い方ができます。if 文としてだけでなく、if 式としても扱うことができます。「式」と書いたとおり、Java の if 文とは違い if 式は評価されて値となります。

リスト 2.14: String を返す if 式 2

```
val result : String = if(true) {"true"} else { "false" }
println(result) // =>true
```

リスト 2.14 のように評価結果の値を受け取り、変数に代入することも可能です。else 節がある場合に、この if は式として扱われます (else 節のない場合は、値を返さない Java と同じような if 文として扱われます)。

条件判定（when 式）

when 式は、Java でいう switch 文のような役割を果たします。if 式と同様に、when 式自体は評価され値となります。

リスト 2.15: when 式 1

```
  val value = 1
  val str = when(value) {
    1-> "one"
    2-> "two"
    else -> "other"
  }
```

条件のうち、必ずどれか 1 つに一致する必要があります。たとえば、リスト

2.15 の場合は 1 か 2 以外の数値の場合もありえるので、else がないとコンパイルエラーとなります。

左側を条件式にすることも可能です。

リスト 2.16: when 式 2

```
val value = 1
when {
  value == 1-> println("one")
  value == 2-> println("two")
  else -> println("other")
}
```

for 文

for 文はリスト 2.17 のように記述します。

リスト 2.17: for 文 1

```
val array: Array<Int>  = arrayOf(1, 2, 3, 4, 5)
for (i in array) {
  print(i) // =>12345
}
```

arrayOf は Kotlin の標準の関数です。この場合は、Array<Int>型のオブジェクトを生成しています。その他にもコレクション生成の関数が揃っています。

- mapOf：マップを生成する関数
- listOf：リストを生成する関数
- setOf：セットを生成する関数

その他にも Range を使った書き方も可能です。

リスト 2.18: for 文 2

```
for (i in 0..4) {
     print(i) // =>01234
}
```

downTo という標準の関数を使うことで、逆順のループも簡単に行うことができます。

リスト 2.19: for 文 3

```
for (i in 4 downTo 0) {
     print(i) // =>43210
}
```

while 文、do-while 文

Java と同じく、while 文や do-while 文も利用可能です。

リスト 2.20: while

```
  var i = 0
  while (i < 10) {
    print(i++) // => 0123456789
  }
```

リスト 2.21: do-while

```
  var i = 0
  do {
    print(i++) // => 0123456789
  } while(i < 10)
```

Null 安全

Null 不許容型と Null 許容型

Kotlin における通常の型は null の代入を許容しません。

リスト 2.22: Null 不許容型

```
var a: String = "abc"
a = null // => コンパイルエラー
```

null を代入するには、型の後ろに?をつけて定義します。

リスト 2.23: Null 許容型

```
var b: String? = "abc"
b = null // => OK
```

このように、コンパイル時に null の代入が許容されるかの判断がなされるため、アプリケーション実行時における null 参照に対するメンバーアクセスによる例外（Java での NullPointerException）を防ぐことができます。

Kotlin において NullPointerException が発生するのは、throw NullPointerException() などのように明示的に発生させる場合か、後述する!!演算子を使用した場合などに限られます[*2]。

上記のように実行時にプログラマーの意図しない箇所で NullPointerException が発生することがほぼないため、Kotlin は Null 安全な言語ということができます。

[*2] null が原因となるエラーを起こす方法は他にもありますが、本書のスコープ外として割愛します

Null チェックと安全な呼び出し

Null 許容型のメソッドやプロパティを参照するとコンパイルエラーとなります。

リスト 2.24: Null 不許容型への参照

```
var a: String = "abc"
val l : Int = a.length // => OK
```

リスト 2.25: Null 許容型への参照

```
var b: String? = "abc"
val l : Int = b.length // => コンパイルエラー
```

Null 許容型のメソッドを呼び出すためには、事前に `if` を用いて `null` でないことをチェックします。

リスト 2.26: Null 許容型のメソッド呼び出し

```
var b: String? = "abc"
val l : Int = if(b != null) b.length else -1
```

?演算子を使えば、Null 許容型のメソッドやプロパティを参照することができます。

リスト 2.27: Null 許容型への安全な参照

```
val l : Int? = b?.length
```

ただし、b が null の場合は l も null となるため、l の型は Null 許容型である Int?になっていることに注意してください。

エルビス演算子

リスト 2.28: Null 許容型のメソッド呼び出し

```
var b: String? = "abc"
val l : Int = if(b != null) b.length else -1
```

リスト 2.28 のような if による null チェックは?:演算子を用いて記述することができます。この演算子を**エルビス演算子**と呼びます。

リスト 2.29: エルビス演算子

```
val l : Int = b?.length ?: -1
```

b が null の場合は l は-1 となり、l の型は Null 不許容型である Int となります。

!!演算子

!!演算子を用いれば、Null 許容型を Null 非許容型に無理やり変換します。

リスト 2.30: !!演算子

```
val l : Int = b!!.length
```

この場合 l は Int 型となります。b が null の場合は実行時に NullPointerException が発生します。つまり!!演算子は危険な操作であ

り、絶対に一生使わないくらいの気持ちでいるとよいでしょう。どうしても Null 非許容型に変換しなくてはならない状況に直面したら、標準関数の `requireNotNull` を使用することを検討してください。

関数

関数は `fun` を使って定義します。

リスト 2.31: 関数名 add、引数 x と y、戻り値の型は Int の関数の定義

```
fun add(x: Int, y: Int): Int {
    return x + y
}
```

引数はリスト 2.32 のように定義します。

リスト 2.32: 関数の引数のフォーマット

```
変数名: 型
```

それぞれの引数はコンマで分けます。すべての引数は、型を明記しなくてはなりません。

戻り値の型はブロックを書いた場合、定義しなくてはなりません。ただし、戻り値の型が `Unit` だった場合はすでに見たとおり、省略することができます。

もし、関数がひとつの式で構成される場合、=を書いてブロックを省略することが可能です。

リスト 2.33: ブロックを省略する

```
fun add(x: Int, y: Int): Int = x + y
```

さらに、コンパイラーが型推論をできるような戻り値の型であれば、戻り値の型の定義を省略することも可能です。

リスト 2.34: 戻り値の型も省略可能

```
fun add(x: Int, y: Int) = x + y
```

関数の呼び出し方法は他の言語と似ています。

リスト 2.35: 関数の呼び出し方

```
val result = add(1, 2)
```

引数のデフォルト値の定義も可能です。型の定義の後に=を書いて定義します。

リスト 2.36: デフォルト値

```
fun toAddFormula(value: Int, num: Int = 1): String {
    return "$value + $num"
}
```

このメソッドを呼び出すにはリスト 2.37 のようにして呼び出します。

リスト 2.37: デフォルト値がある関数の呼び出し方

```
val result1 = toAddFormula(10) // "10 + 1"
val result2 = toAddFormula(15, 2) // "15 + 2"
```

リスト 2.38 は名前付き引数を利用した例です。名前付き引数は引数の順番を変えても問題なく動きます。名前付き引数を使った場合、後に続く引数はすべて

名前付き引数にしなくてはなりません。

リスト 2.38: 名前付き引数を使っての呼び出し方

```
val result3 = toAddFormula(value = 100) // "100 + 1"
val result4 = toAddFormula(num = 3, value = 1234) // "1234 + 3"
val result5 = toAddFormula(1000, num = 5) // "1000 + 5"
val result6 = toAddFormula(num = 4, 101) // コンパイルエラー
val result6 = toAddFormula(value = 12, 7) // コンパイルエラー
```

この機能の良いところは、たとえばリスト 2.39 のような複数の同じ型のメソッドを呼び出す場合、リスト 2.40 のように効力を発揮します。

リスト 2.39: 複数の同じ型のメソッド

```
fun displayName(firstName: String,
                lastName: String,
                middleName: String) {
}
```

リスト 2.40: 複数の同じ型のメソッド

```
displayName("名前", "姓", "ミドルネーム")
// コンパイルは通るが挙動的に NG
displayName("名", "ミドルネーム", "姓")
// 明示的にすることにより間違いを減らすことが可能
displayName(lastName = "ことり", firstName = "ん", middleName = "K")
```

関数は型パラメータを扱うことも可能です。関数名の前に<>を使って宣言します。

リスト 2.41: 型変数付き関数

```
    fun <T> singletonList(item: T): List<T> {
    }
```

高階関数とラムダ式

Kotlin では高階関数やラムダ式を扱うことができます。具体的な文法を見ていきましょう。

高階関数

高階関数とは、関数オブジェクトを引数にしたり、戻り値にする関数です。リスト 2.42 に示す関数 `lock` は高階関数の例です。第二引数の `body: () -> T` は、引数なしで戻り値に任意の型を取る関数を表します。この例では `body` という具体的な処理を外に出すことによって、`lock` という関数名のとおりロックすることを目的とした関数を宣言し、汎用性が高くなります。

リスト 2.42: 高階関数の例

```
// body という名前の引数に関数オブジェクトを渡している
fun <T> lock(lock: Lock, body: () -> T): T {
  lock.lock()
  try {
    // 引数で渡された関数オブジェクトを実行している
    return body()
  }
  finally {
    lock.unlock()
  }
}
```

ラムダ式

ラムダ式を使うと、関数を宣言せずに関数オブジェクトをすぐに生成することができます。リスト 2.43 は高階関数の例ですが、第二引数に注目してください。第二引数にはラムダ式で a,b という 2 つの引数をとり、比較結果を返すという関数オブジェクトが記述されています。

リスト 2.43: ラムダ式の例

```
max(strings, { a, b -> a.length < b.length })
```

また、関数の引数の最後にラムダ式を渡している場合は、引数の外に出して、リスト 2.44 のように記述することが可能です。ラムダ式内に複数行記述する場合でも、簡潔に可読性の高いまま関数を記述することができます。

リスト 2.44: ラムダ式の外出し

```
max(strings) { a, b -> a.length < b.length }
```

インライン関数

実行時に関数オブジェクトを生成することは、メモリアロケーション等コストが高くなる場合があります。この問題を解決するために、インライン関数という仕組みが用意されています。これを使用すると、コンパイル時に関数がインライン展開されます。

具体例を見ていきましょう。リスト 2.42 の関数の使用は次のようになります。

リスト 2.45: 関数の使用例

```
lock(l) { foo() }
```

この関数をインライン関数にするには次のように inline を付けるだけです。

リスト 2.46: インライン関数

```
inline fun lock<T>(lock: Lock, body: () -> T): T {
  // ...
}
```

インライン展開されたコードは次のようになりますが、lock の使用者はこれを意識する必要はありません。

リスト 2.47: インライン展開されたコード

```
l.lock()
try {
  foo()
}
finally {
  l.unlock()
}
```

クラス

Kotlin は Java 同様オブジェクト指向型の言語です。

クラスとインスタンス

クラスは class キーワードで定義することができます。

リスト 2.48: MyClass.kt

```
class MyClass {
}
```

クラスにメンバ（プロパティやメソッド）がない場合、波括弧は省略できます。リスト 2.49 のコードはリスト 2.48 のコードと同じ意味となります。

リスト 2.49: 波括弧を省略した MyClass.kt

```
class MyClass
```

定義したクラスはリスト 2.50 のように記述することで、インスタンスを生成することができます。Java のような new キーワードは不要です。

リスト 2.50: インスタンス生成

```
val myClass = MyClass()
```

コンストラクターとイニシャライザー

先ほど紹介した MyClass ではコンストラクターを記述していませんが、この場合デフォルトコンストラクターが自動で生成されます。明示的にコンストラクターを定義する場合、リスト 2.51 のように記述します。

リスト 2.51: コンストラクター

```
class Person(name: String) {
}
```

リスト 2.51 のように、クラス名の後ろに定義するコンストラクターを**プライマ**

リーコンストラクターと呼びます。

アノテーションや可視性修飾子を付ける場合は、リスト 2.52 のように constructor キーワードが必要となります。

リスト 2.52: constructor キーワード付きのプライマリーコンストラクター

```
class Person public @Inject constructor(name: String) {
    // 処理
}
```

プライマリーコンストラクターには処理を記述することができません。インスタンス生成時に行いたい初期化処理はイニシャライザーに記述します。イニシャライザーは init キーワードで定義します。

リスト 2.53: イニシャライザー

```
class Person(name: String) {
    init {
        logger.info("name = ${name}")
    }
}
```

コンストラクターを複数定義したい場合、**セカンダリーコンストラクター**を定義する必要があります。セカンダリーコンストラクターは constructor キーワードを利用してリスト 2.54 のように記述します。また、次のコードでは this キーワードを使用しプライマリーコンストラクターを呼び出しています。

リスト 2.54: セカンダリーコンストラクター

```
class Person(val name: String) {
    constructor(name: String, parent: Person) : this(name) {
        // 処理
    }
}
```

リスト 2.54 のコードでは、プライマリーコンストラクターの引数に val キーワードを用いてプロパティを作成していますが、プロパティについては後ほど説明します。

メソッド

クラス内に関数を記述することでメソッドを定義することができます。

リスト 2.55: メソッド

```
class Greeter {
    fun greet(name: String) {
        println("Hello, $name!")
    }
}
```

リスト 2.56 のように呼び出すことができます。

リスト 2.56: メソッドの呼び出し

```
val greeter = Greeter()
greeter.greet("Kotlin")  // 「Hello, Kotlin!」と出力される
```

プロパティ

プロパティはリスト 2.57 のように var, val キーワードで記述することができます。var は変更可能プロパティ、val は変更不可能な読み取り専用プロパティです。

リスト 2.57: プロパティ

```
class Person {
    var name: String = ""
}
```

上記リスト 2.57 のコード相当の処理を Java で書くと、リスト 2.58 のようになります。

リスト 2.58: プロパティを Java で記述した場合

```
final class Person {
   private String name;
   public String getName() {
       return name;
   }
   public String setName(String name) {
       this.name = name;
   }
}
```

Kotlin のプロパティには、自動的にバッキングフィールドと呼ばれるものが生成され、実際の値はそこに格納されます。要は `var`, `val` キーワードでフィールドのように記述をするだけで、getter, setter 付きのフィールドが自動生成されるということです（`val` は読み取り専用なので getter のみ）。

プロパティにアクセスするには、リスト 2.59 のようにフィールドを呼び出す感覚で記述します。

リスト 2.59: プロパティへのアクセス

```
val taro = Person()
taro.name = "たろう"
println(taro.name)  // 「たろう」と出力
```

自分で getter と setter を定義することもできます。これをカスタムゲッター、カスタムセッターと呼びます。カスタムゲッターを定義した場合はバッキングフィールドは生成されません。

リスト 2.60: カスタムゲッター・カスタムセッター

```
    var stringRepresentation: String
        get() = this.toString()
        set(value) {
            setDataFromString(value) // 文字列をパースして他のプロパティへ
値を代入する
        }
```

lateinit キーワード

プロパティは必ず初期化する必要があります。しかし、**DI**（Dependency Injection）[*3]を利用している場合などは、インスタンス生成時に値を設定することはできません。そういった場合、`lateinit` キーワードで初期化を遅らせることができます。

リスト 2.61: lateinit

```
    class MyClass {
        lateinit var foo: String
    }
```

`lateinit` は `var` でしか利用できないため、初期化後に変更される可能性があります。また、初期化される前にアクセスすると例外が発生します。取扱には十分に注意しましょう。

[*3] デザインパターン（設計思想）の一種。Android で DI を利用したい場合、Dagger2 というライブラリーが有名です

オブジェクト

class の代わりに object キーワードを使うことにより、インスタンスが必ず 1 つしか生成されない、シングルトンなクラスを定義できます。

リスト 2.62: オブジェクト

```
object DataProviderManager {
  fun registerDataProvider(provider: DataProvider) {
      // ...
  }
}
```

オブジェクトに定義したメソッドを呼び出すには、インスタンスの生成処理を記述せずクラス名から直接メソッドを参照します。

リスト 2.63: オブジェクトの参照

```
DataProviderManager.registerDataProvider(...)
```

コンパニオンオブジェクト

Kotlin では Java のようにクラスに static なメンバを定義することができません。static のようにインスタンス生成をしなくても利用できるメンバを定義するためには、コンパニオンオブジェクトを定義します。

リスト 2.64: コンパニオンオブジェクト

```
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}
```

コンパニオンオブジェクトに定義したメンバを使用するには、次のようにクラスから直接参照します。

リスト 2.65: コンパニオンオブジェクトのメンバ参照

```
val instance = MyClass.create()
```

継承

open 修飾子がついているクラスについては:で継承することができます。

リスト 2.66: 継承

```
open class Person(val name: String)
class Student(name: String, val id: Long): Person(name)
```

この場合、Person がスーパークラスで、Student がサブクラスとなります。スーパークラスに open がついていない場合はコンパイルエラーとなるので注意が必要です。多重継承はできないため、スーパークラスは 1 つしか定義することができません。また、リスト 2.66 のコード例では Student で受け取った引数 name を、Person のプライマリーコンストラクターに渡しています。

また、スーパークラスの open 修飾子がついているメンバをサブクラスの同じ名前のメンバでオーバーライド（上書き）できます。上書きする側のメンバには override 修飾子をつけます。

リスト 2.67: オーバーライド

```
open class Person(val name: String) {
    open fun introduceMyself() {
        println("I am $name.")
    }
}

class Student(name: String, val id: Long): Person(name) {
    override fun introduceMyself() {
        println("I am $name. (id=$id) ")
    }
}
```

これを実行すると、リスト 2.68 の結果になります。

リスト 2.68: オーバーライドの実行例

```
val student: Student = Student("きの子", 728)
student.introduceMyself()   // 「I am きの子. (id=728)」を出力
```

インターフェース

interface キーワードで、インターフェースを定義することができます。

リスト 2.69: インターフェース

```
interface MyInterface {
    fun bar()
    fun foo() {
      // optional body
    }
}
```

リスト 2.70 のように実装することができます。

リスト 2.70: インターフェースの実装

```
    class Child : MyInterface {
        override fun bar() {
            // body
        }
    }
```

MyInterface においてメソッド bar は処理が未定義であるため、Child でオーバーライドして処理を実装する必要があります。foo メソッドは MyInterface ですでに処理が実装されているため、改めて実装する必要はありません。

この Child のインスタンスの挙動はリスト 2.71 のようになります。

リスト 2.71: インターフェースを実装したクラスの実行結果

```
    val child = Child()
    child.bar()  // Child のオーバーライドした bar メソッドで定義されている処理が実行される
    child.foo()  // MyInterface の foo メソッドで定義されている処理が実行される
```

継承と異なり、インターフェースはカンマで区切ることにより複数実装が可能です。また、インターフェースにインターフェースを継承することもできます。

リスト 2.72: 複数のインターフェースの実装例

```
interface A
interface B
interface C: B  // インターフェース C にインターフェース B を継承する

class D: A, C   // クラス D にインターフェース A とインターフェース C を実装する
```

インターフェースの匿名クラスを生成したい場合、オブジェクト式を利用する

とインターフェースから匿名クラスを作成することができます。オブジェクト式は object: を用いて記述します。

リスト 2.73: オブジェクト式による匿名クラス生成

```
interface Greeter {
    fun greet()
}

val greeter = object: Greeter {
    override fun greet() {
        println("Hello")
    }
}
```

パッケージ

Kotlin では、Java と同様パッケージによりクラスなどの要素を名前空間で区切ることができます。パッケージを定義するには、ファイル先頭で package キーワードを使用します。

リスト 2.74: パッケージの宣言

```
package sample.hoge
class Foo
```

これで Foo クラスは sample.hoge パッケージに属すことになり、異なるパッケージからは sample.hoge.Foo という名前でアクセスする必要があります。Java 同様 import キーワードでインポートすることで、単純なクラス名でアクセスすることができます。

リスト 2.75: パッケージの宣言

```
package sample.fuga
import sample.hoge.Foo

class Bar {
    fun doSomethingGood() {
        Foo()  // インポートしているため、クラス名のみでアクセスできている
    }
}
```

アクセス制限

可視性修飾子をつけることにより、パッケージのトップレベルに属する関数やクラスなどの公開範囲を設定することができます。表 2.1 は「Kotlin スタートブック」[6] より引用した、トップレベルにおける可視性修飾子の一覧です。

表 2.1　トップレベルにおける可視性修飾子

修飾子	公開範囲
public(デフォルト)	公開範囲に制限はなく、どこからでもアクセス可能です
internal	同一モジュール内に限り、全公開です
private	同一ファイル内のみ、アクセス可能です

`internal` の「同一モジュール」というのは、コンパイル単位、たとえば Maven や Gradle の 1 つのプロジェクトを指します。

クラス内のメンバについても可視性修飾子をつけることができます。表 2.2 は「Kotlin スタートブック」[6] より引用した、クラスにおける可視性修飾子の一覧です。

データクラス

Android 開発だけに限りませんが、情報を保持するためだけのクラスをよく作成すると思います。こういったクラスでは保持するメンバーがわかりさえすれ

表 2.2　クラスにおける可視性修飾子

修飾子	公開範囲
public(デフォルト)	公開範囲に制限はなく、どこからでもアクセス可能です
internal	同一モジュール内に限り、全公開です
protected	同一クラス内と、サブクラス内からアクセス可能です
private	同一クラス内のみ、アクセス可能です

ば、getter、setter や `toString()` などのメソッドは標準的なものを生成することが可能です。Kotlin では `data` クラスとしてこのような機能を提供しています。

たとえば `id`、`name` を持った `Person` クラスを定義してみます。

リスト 2.76: Person.kt

```
data class Person(val id: Long, var name: String)
```

`data` クラスを定義するとコンパイラーが次のメソッドを自動で生成します。

- `equals()`/`hashCode()`
- `toString()`
  - `Person(id=100, name=Tarou)` 形式
- `componentN()`
  - N は 1 から開始されるメンバー変数の順番。たとえば `Person` クラスの `component2()` の場合、`name` の値が戻り値となる
- `copy()`
  - 特定のメンバー変数が同じのインスタンスを作成したい時に便利なメソッド（リスト 2.77）

リスト 2.77: copy() の実装イメージ

```
    fun copy(id: Long = this.id, name: String = this.name): Person {
        return Person(id, name)
    }
```

そのため、リスト 2.76 で定義した Person クラスはメソッドを書いていませんが、リスト 2.78 のようにメソッドの呼び出しが可能です。

リスト 2.78: data class 利用例

```
    val person = Person(8000, "しらじ")
    println(person.name) // しらじ
    println(person.toString()) // Person(id=8000, name=しらじ)
    val person2 = person.copy(id = 8001)
    println(person2.toString()) // Person(id=8001, name=しらじ)
    person2.name = "しらじのにせもの" // name は var のため上書き可。id は val
のため上書き出来ない
    println(person2.toString()) // Person(id=8001, name=しらじのにせもの)
    println(person == person2) // false
    val person3 = Person(8000, "しらじ")
    println(person == person3) // true
```

data クラスは素晴らしいですが、いくつか制限もあります。

- Primary コンストラクターは少なくとも 1 つの引数をもつ必要がある
- Primary コンストラクターのすべての引数は val か var で定義する必要がある
- data クラスは abstract、open、sealed、inner はつけられません
- 継承できるのは sealed クラスのみ。Interface の実装もできる

data クラスは**分解宣言**（Destructuring Declarations）もサポートしています（リスト 2.79）。分解宣言とはインスタンスを分解して一度にいくつかの変数に代入できる機能です。

リスト 2.79: 分解宣言

```
val person = Person(1, "二郎")
val (id, name) = person
println("id: $id - name: $name") // "id: 1 - name: 二郎"と表示される
```

Extension

拡張関数（Extension Function）

拡張関数とは任意の型（クラスやインターフェースなど）に対して外部から関数を追加できる機能です。たとえば String などのよく使われるクラスにこんな関数があったら良いのに…と思ったことはありませんか？ 拡張関数ならそれが可能です。

リスト 2.80: 拡張関数の追加

```
fun String.appendBeer() : String = "${this}beer!"
```

リスト 2.80 は拡張関数の実際の例です。関数を追加したい型を関数名の手前に付け加えるだけです。その他は普通の関数の書き方と違いはありません。

リスト 2.81: 拡張関数の呼び出し

```
println("I like".appendBeer()) // => I like beer!
```

リスト 2.81 のように呼び出します。拡張関数は自由に型に対して関数を追加することが可能なので、確かに便利なものなのですが乱用は禁物です。リスト 2.80 のように有用でない（局所的にしか利用しない）関数などが増えると混乱を招きます。ボイラープレートを減らせる、生産性に寄与するような、ここぞという時のために定義するのを推奨します。

拡張プロパティ（Extension Properties）

拡張プロパティも拡張関数と考え方は同じです。任意のクラスに対して外部からプロパティを追加できる機能です。

リスト 2.82: 拡張プロパティ

```
val <T> List<T>.lastIndex: Int
    get() = size - 1
```

リスト 2.82 は Kotlin に標準で実装されているものです。List インターフェースに対してプロパティを追加しています。

リスト 2.83: 拡張プロパティの呼び出し

```
    val arr = listOf(1,2,3)
    println(arr.lastIndex) //=> 2
```

リスト 2.82 のように呼び出します。リストの最後のインデックスである 2 が標準出力されます。メリット・デメリットは拡張関数と同様ですので割愛します。

拡張関数・拡張プロパティのスコープ

リスト 2.84: 拡張関数と拡張プロパティを定義したパッケージ

```
    package a

    fun String.extentionFun() = "it's cool operation"
    val String.extentionProperty : String
    get() = "something special"
```

リスト 2.84 のように `package a` で、拡張関数と拡張プロパティを宣言した

場合、

リスト 2.85: 拡張関数と拡張プロパティの別パッケージからの呼び出し

```
package b

import a.extentionFun
import a.extentionProperty

fun main(args : Array<String>) {
    "hoge".extentionFun()
    "fuga".extentionProperty
}
```

リスト 2.85 のように別パッケージから拡張関数・拡張プロパティを呼び出すには `import` が必要です。同一パッケージ内であれば `import` 無しで拡張関数や拡張プロパティが呼び出せます。たまに見る使われ方は、拡張関数のみ集めた kt ファイルを作成し、拡張関数用の package を作成しそれを明示的に import させるような手法です。そうすることで、スコープを分けられ拡張関数を管理しやすくなるようです。

その他

Kotlin には、面白く便利な機能がまだまだあります。それらを簡単に紹介して、この節を締めくくりたいと思います。

演算子オーバーロード

プログラミング言語で使用できる演算子を、新しい型にも適用できるようにする仕組みを**演算子オーバーロード**と言います。Kotlin では、演算子による計算は、実際にはメソッドや拡張関数への呼び出しになる方式でこれを実現しています。使用できる演算子の種類はあらかじめ決められており、各演算子は決まったシグネチャーのメソッド・拡張関数に対応します。たとえば、`plus` というメソッドが定義されていれば、`+` という 2 項演算子を使った計算を行えるわけです。

リスト 2.86: 演算子オーバーロード

```
class MyInt(val value: Int) {
    operator fun plus(that: MyInt): MyInt =
        MyInt(value + that.value)
}

fun main(args: Array<String>) {
    val sum: MyInt = MyInt(5) + MyInt(7)
    println(sum.value) // 12
}
```

リスト 2.86 では、独自に定義したクラス MyInt にメソッド plus を持たせています。修飾子 operator が重要で、これをメソッドや関数に付けることで、対応する演算子のオーバーロードが行えます。このメソッド plus の存在により MyInt(5).plus(MyInt(7)) と呼び出すべきところを、MyInt(5) + MyInt(7) と呼び出すことも可能になります。

演算子とそれに対応するメソッドは、公式サイト[*4]で確認できます。

イコール

Java の参照型における==による比較は、Kotlin では===で比較します。この比較は、同じ参照であるかどうかを比べているのはご存知かと思います。そのため、Java ではオブジェクトの中身が同じかどうかを調べる際には equals メソッドを使います。一方 Kotlin では、オブジェクトの中身が同じかどうかを==で調べることができます。==も演算子オーバーロードのひとつであり、すなわちシンタックスシュガーとなっています。a == b の式は a?.equals(b) ?: (b === null) と同じです。どういうことかというと、a が null でない場合は、b を引数にメソッド equals を呼び出し、a が null である場合は、b も null なのかをテストしています。

[*4] http://kotlinlang.org/docs/reference/operator-overloading.html

型エイリアス

型エイリアス（type alias）という機能を使うことで、既存の型に別名を与えることができます。別名を与えることで、用途がわかりやすくなったり、コードが読みやすくなったりします。

リスト 2.87: 型エイリアスの例

```
typealias Name = String

fun hello(name: Name) {
    println("Hello, $name!")
}

fun main(args: Array<String>) {
    hello("Kotlin")
}
```

リスト 2.87 でキーワード `typealias` を用いて型 `String` に `Name` という別名を与えています。この宣言で `String` の代わりに `Name` を使えるようになります。あくまで別名であり `Name` もまた `String` であることに注意をしてください。そのため、リスト 2.87 のような使い方はイマイチかもしれません[*5]。

関数型に対して適切に別名を付ける使い方がお勧めです。はい、関数にも型があるのです。たとえば `(T) -> Boolean` という書き方をしますが、これは「任意の型 `T` を引数に取って、`Boolean` を返す関数」と読めます。このとき、短く用途の分かる名前を付けてやると可読性が上がるでしょう（リスト 2.88）。

リスト 2.88: 関数型に別名を与える

```
typealias Predicate<T> = (T) -> Boolean
```

[*5] この例では `Name` という新しい型を定義した方がよいかもしれません。そうすることで、名前でない文字列をうっかり `Name` 型の変数に代入するという事故を防ぐことができます

2.3 Java から Kotlin への移行

本書を読んでいる方のほとんどは、普段は Java を使って Android アプリケーションを開発していると思います。Kotlin の学習を進めると同時に、現在の Android アプリケーションにどのように Kotlin を適用していくかを考えていかなければなりません。本節では Java で書いている Android アプリケーションをどのように Kotlin に移行していくかを論じます。

相互互換性のある Kotlin と Java

Kotlin と Java は相互に互換性があります。つまり Kotlin から Java のクラスや関数を利用したり、Java から Kotlin のクラスや関数を利用できます。Java と Kotlin が混在した状態で開発できるので、Java で書いている Android アプリケーションに、あとから Kotlin を導入することは難しくありません。もちろん Kotlin から Java を呼び出すとき、Java から Kotlin を呼び出すときでそれぞれ意識しなければならないことがあります。それらにかかるコストも鑑みて移行を考えなければなりません。

このことから Kotlin に移行していくに当たって次のアプローチが考えられます。

- 混在させる
- 境界を設ける
- 置き換える

これらは学習と適用のフェーズをどのように進めるのかという戦略の話になります。それぞれにいくつかの具体的な方法があり、メリットとデメリットがあります。

混在させる

まずは単純に Java と Kotlin を混在させながら、徐々に Kotlin に移行していくケースを考えてみます。

新しいコードを Kotlin にする

これは今後の追加や変更の一部あるいはすべてを Kotlin で書くというアプローチです。

■メリット 追加や変更のほとんどを Kotlin で行うことになるので、チームは常に Kotlin に触れ続けることになります。Kotlin でどのように書くかについて考えたり議論したりしやすいので、学習の効率は良いでしょう。また、プロダクションコードにおいて Kotlin の価値をすぐに享受でき、変更すればするほど Kotlin 化が進んで移行の負担が下がっていきます。

■デメリット 変更対象を Kotlin 化することで影響を受ける箇所が多数存在する場合、コンフリクトのリスクが高まり並行作業に影響が出やすいです。また Java との連携部分についても意識を向ける必要があるため、壊れないようにメンテナンスするコストが増大します。レビューにおいて Kotlin と Java の両方を読むことになるのでレビュアーの負担は増大します。また Kotlin 化を進めるなかで複雑過ぎて Kotlin 化できない領域が発生し、完全な Kotlin 化になかなか至らない可能性が高いです。

混在させるアプローチをとるのは正直かなり危険だと筆者は考えます。新規コードではなく、単純な POJO などから置き換えるということも考えられますが、これも結局 Kotlin 化されない部分が発生してメンテナンスコストが増大したままになるリスクがあります。

境界を設ける

次にテストやライブラリーなど、Kotlin で書く部分と Java で書く部分の境界を設けるアプローチを考えてみます。

テストから導入する

プロジェクトのテストコードを Kotlin で書くというアプローチです。テストコードを一気に全部 Kotlin にしても、徐々に導入しても構いません。

■メリット プロダクションコードに Kotlin を含めないので、もし壊れたとしても安全です。大胆に導入が可能となり、学習の効率は良いでしょう。テストコードにおいて Kotlin の価値をすぐに享受できます。

■デメリット プロジェクト全体で見ると Java と Kotlin が混在していることに変わりはなく、レビューの負担は高まります。また、テストとプロダクションコードで Kotlin と Java をスイッチして考える必要があるので負担が高く、だんだんテストを書かなくなってしまうといったリスクが考えられます。

いきなり混在して書くアプローチよりは大分まともです。テストの他に分割したモジュールを Kotlin で書くといったことも考えられます。どちらかというと学習が主な目的となり、置き換えは別のアプローチを検討することになります。

ライブラリーを Kotlin で書いて導入する

プロダクトで使うライブラリーを Kotlin で書いて導入するアプローチです。

■メリット 独立して開発をするので、テストから導入するケースに比べてさらに自由度が増します。プロジェクト全体が Kotlin なので、開発中は Kotlin に集中できます。

■デメリット ライブラリー化すべきテーマを適切に探すのは簡単ではありません。ライブラリーを利用するプロダクションコード側は Java なので、Java から Kotlin を利用する際の注意点に対処する必要があります。ライブラリーが完成するまでにある程度時間がかかるので、プロダクション側で恩恵を受ける距離は少し遠いです。ライブラリーを一部の人が書くといった状態になると学習に偏りが生じてしまいます。

テーマさえ見つかれば、ライブラリーを Kotlin で作るアプローチは学習と実益を兼ねていてとても良いと思います。

置き換える

最後に一気にプロジェクト全体を Kotlin に置き換えるアプローチを考えてみます。

プロジェクト全体を一気に Kotlin に置き換える

実は Kotlin プラグインが提供する、Java を Kotlin に変換する機能はプロジェクト全体にも適用できます。プロジェクトのファイルツリーでディレクトリーを選択した状態で Kotlin への変換を実行すると、そのディレクトリー配下のすべての Java ファイルを Kotlin に変換できます。ただしこの方法はあまりうまくいくことはありません。Java から Kotlin の変換機能は、あくまで Java の文法を Kotlin に置き換えるに過ぎず、適切な設計になるわけではないからです。ほとんどの場合、Null 許容型の境界の問題がアプリケーション全体に氾濫し、収拾がつかなくなるでしょう。ここでは変換機能を活用しつつ全体を手動で置き換えていくケースを想定します。

■メリット 他の方法に比べると最短で Kotlin への移行ができます。Java と混在するケースのように Java から Kotlin を呼び出すケースを考える必要がないので、その分のメンテナンスコストはなくなります。

■デメリット 学習を想定していないので、チームがすでに Kotlin を使いこなせるか、圧倒的にリードする人間が必要になります。

現実的には、テストやライブラリーに Kotlin を導入して学習をし、その後一気にプロダクションコードを置き換えるという戦略が一番コストが低いのではないかと思います。

Kotlin から Java を利用する

プロジェクトを Kotlin に一気に置き換えるにせよ、部分的に置き換えるにせよ、既存の Java ライブラリーを利用するときに Kotlin と Java の境界を意識することになります。本項では、Kotlin から Java を利用するときに意識する必要

がある点について解説します。

識別子をエスケープする

Java では予約語ではないが、Kotlin では予約語というキーワードがいくつか存在します。`in`、`object`、`is` などです。

Java のクラスがこうした Kotlin の予約語と衝突するようなフィールドや関数を持っていた場合、次のようにバッククォートでエスケープして呼び出します。

リスト 2.89: Kotlin の予約語をエスケープする

```
foo.‘is‘(bar)
```

null とプラットフォーム型

Java には Null 許容型という概念がありません。Java の参照型は常に null になる可能性があります。この性質は Kotlin から制御することはできないので、Java 側で宣言しているフィールドや関数の戻り値の型は、プラットフォーム型として特別な扱いを受けます。具体的には null チェックが緩和されて、Java と同じ様なふるまいになります。

たとえば Activity の findViewById 関数を例に考えてみましょう。Activity の findViewById 関数は Java の関数なので、戻り値はプラットフォーム型となります。プラットフォーム型は Null 許容型か非 Null 型かが曖昧なため、リスト 2.90 のように利用者が宣言時に決めることになります。

リスト 2.90: findViewById 関数とプラットフォーム型

```
val text = findViewById(R.id.text) as TextView
text.id

val text2 = findViewById(R.id.text) as TextView?
text2?.id
```

もし findViewById 関数が View を返せば両者は正しく動作します。null を返せば TextView で宣言した方は、`text.id` の箇所で NullPointerException をスローします。

この性質は厄介です。実行しなければプラットフォーム型が実際に null になるかどうか分からないからです。こうした曖昧な宣言を回避するために、`org.jetbrains.annotations.Nullable` アノテーションと `org.jetbrains.annotations.NotNull` アノテーションが用意されています。Java 側でこれらのアノテーションが適切に宣言されていれば、プラットフォーム型になることを回避できます。

検査例外の取扱い

Kotlin は Java の検査例外をリスト 2.91 のように無視します。

リスト 2.91: 検査例外を無視する

```
val file = File("not found")
val inputStream = file.inputStream() // FileNotFoundException
```

もちろん例外がスローされないわけではないので、適宜 try-catch 式を書くことになります。

Class<T>型を参照する

Kotlin から Class<T>型を参照するにはリスト 2.92 のように書きます。

リスト 2.92: Class 型を参照する

```
// クラスから取り出す
val clazz : Class<File> = File::class.java

// インスタンスから取り出す
```

```
val file = File(".")
val clazz : Class<File> = file.javaClass
```

ラムダ式と SAM インターフェースの変換

Kotlin は Java の関数の引数が **SAM インターフェース**（Single Abstract Method Interface）の場合、ラムダ式を自動的に SAM インターフェースに変換してくれます。

たとえば Thread クラスはコンストラクターで Runnable インターフェースを受け取ります。Runnable インタフェースは run 関数をひとつだけもつ SAM インターフェースです。SAM インターフェースの自動変換によってリスト 2.93 のように、ラムダ式で Runnable の実装を渡せます。

リスト 2.93: SAM インターフェースの変換の例

```
val thread = Thread {
  Log.d("kotlin", "hello")
}
```

この性質は Java の SAM インターフェースに対してのみ動作する点に注意する必要があります。SAM インターフェースだとしても Kotlin で宣言しているインターフェースの場合、自動変換の対象にはなりません。

Java から Kotlin を利用する

Java と Kotlin を混在させるケースや、ライブラリーを Kotlin で書いて導入するケースでは、Java から Kotlin を利用する場合への対処が必要になります。本項では、Java から Kotlin を利用するときに知っておくと良い点について解説します。

プロパティにアクセスする

Kotlin のプロパティには暗黙的に getter、setter が実装されています。Java から Kotlin のクラスのプロパティへのアクセスは、リスト 2.94 のように getter、setter を通して行います。

リスト 2.94: プロパティへのアクセス

```
// Kotlin
class User(val id: Long, var name: String)

// Java
User user = new User(1L, "taro");
user.getId(); // 1L
user.getName();  // "taro"
user.setName("jiro");
```

プロパティのフィールドに直接アクセスする

プロパティの実体であるバッキングフィールドは、プロパティの可視性にかかわらず常に private で宣言されているので、直接触れることはできません。プロパティのバッキングフィールドに直接触れたいという場合は、`@JvmField` アノテーションを使います。

リスト 2.95 のようにプロパティの宣言に`@JvmField` アノテーションを付与すると、getter、setter を生成する代わりにバッキングフィールドを public で宣言します。

リスト 2.95: @JvmField アノテーションをプロパティに設定する

```
// Kotlin
class User(@JvmField val id: Long, var name: String)

// Java
User user = new User(1L, "taro");
```

```
user.id; // 1L
```

プロパティに@JvmField アノテーションを付与した場合、プロパティに対するカスタムアクセサーは定義できなくなります。また、アノテーションを付与するには、private、open、const、override ではなく、さらにデリゲートを使用していないという条件があります。

パッケージレベルの関数

Kotlin はリスト 2.96 のようにパッケージレベルで関数やプロパティを定義できます。

リスト 2.96: パッケージレベルで関数やプロパティを定義する

```
// Config.kt
package org.kotlinlang

val CLIENT_ID = "aaaa"
fun hello(){ /* ... */ }
```

これらの関数やプロパティはクラスに属していないように見えますが、実態としては Kotlin ファイルの名前に接尾辞 Kt を付与したクラスの中に静的に宣言されます。Java からパッケージレベルの関数やプロパティを使うにはリスト 2.97 のように書きます。

リスト 2.97: パッケージレベルの関数やプロパティを使う

```
ConfigKt.getCLIENT_ID(); // "aaaa"
ConfigKt.hello();
```

オブジェクトやコンパニオンオブジェクトの関数やプロパティを静的にする

オブジェクトやコンパニオンオブジェクトはそれぞれシングルトンのインスタンスが宣言されます。Kotlin 上ではインスタンスへのアクセスを省略して静的にアクセスしているように書けますが、Java からはリスト 2.98 のように実際のインスタンスを経由しなければなりません。

リスト 2.98: オブジェクトへのアクセス

```
// オブジェクト
// INSTANCE を経由する必要がある
DataProviderManager.INSTANCE.registerDataProvider(this);

// コンパニオンオブジェクト
// Companion を経由する必要がある
MainActivity.Companion.createIntent(context);
```

オブジェクトやコンパニオンオブジェクトの、関数やプロパティに@JvmStatic アノテーションを付与することで、Java 上で静的に宣言できます（リスト 2.99）。

リスト 2.99: @JvmStatic アノテーションを付与する

```
object DataProviderManager {
  @JvmStatic
  fun registerDataProvider(provider: DataProvider) {/* ... */}
}
class MainActivity : AppCompatActivity() {
  companion object {
    @JvmStatic
    fun createIntent(context: Context): Intent {
      //...
    }
  }
}
```

リスト 2.100 のようにスッキリと記述できるようになります。

リスト 2.100: オブジェクトやコンパニオンオブジェクトに静的にアクセスする

```
DataProviderManager.registerDataProvider(this);
MainActivity.createIntent(context);
```

拡張関数を呼び出す

たとえばリスト 2.101 のような拡張関数があるとします。

リスト 2.101: IntExtensions.kt

```
package org.kotlinlang
fun Int.reversed(): Int {
    return this.toString().reversed().toInt()
}
```

Kotlin からはリスト 2.102 のように拡張関数を呼び出します。

リスト 2.102: Kotlin で拡張関数を使う

```
234.reversed() // 432
```

Kotlin 上では拡張関数の対象のクラスに関数が増えたように見えますが、実際はリスト 2.103 のように、対象のクラスを第一引数に取る関数が宣言されます。

リスト 2.103: IntExtensionsKt クラス

```
package org.kotlinlang
public final class IntExtensionsKt {
```

```
   public static final int reversed(int $receiver) {
     // ...
   }
}
```

ですので、Java から拡張関数を呼び出す時はリスト 2.104 のように書きます。

リスト 2.104: Int.reversed() を使う

```
IntExtensionsKt.reversed(234); // 432
```

関数のオーバーロード

関数がデフォルト引数を持っているとき、Kotlin 上ではその関数がオーバーロードしているように見えます（リスト 2.105）。

リスト 2.105: デフォルト引数を持つ関数

```
class UserRepository {
  fun get(id: Int, param: String = ""): User {
    return User(1, "")
  }
}

val repository = UserRepository()

// オーバーロードしているように見える
repository.get(10)
repository.get(10, "page=10")
```

しかし Java からは引数をフルセットで渡す必要があります（リスト 2.106）。

リスト 2.106: Java からは常に引数がフルセット必要になる

```
UserRepository repository = new UserRepository();
repository.get(10, "page=10");
```

デフォルト引数をもつ Kotlin の関数をオーバーロードさせるには、`@JvmOverloads` アノテーションを使います（リスト 2.107）。

リスト 2.107: @JvmOverloads アノテーションを使う

```
class UserRepository {
  @JvmOverloads
  fun get(id: Int, param: String = ""): User {
    return User(1, "")
  }
}
```

これで実際にオーバーロードした関数を Java から呼び出せます（リスト 2.108）。

リスト 2.108: オーバーロードした関数を使う

```
UserRepository repository = new UserRepository();
repository.get(10); // OK
repository.get(10, "page=10");
```

本当に移行するべきなのかどうか

本節では Java で書いている Android アプリケーションを Kotlin に移行するためのアプローチや、Kotlin と Java を相互に利用する際に知っておくとよい知識について解説しました。

Java から Kotlin へ移行することによって得られる恩恵は、プロダクトやチー

ムや環境ごとに異なります。本節が移行のために必要なコストの試算や本当に移行するべきかどうかの判断の一助になれば幸いです。

第 3 章

次のステップ

3.1　学習方法

この節では、Kotlin のはじめの一歩を踏み出すための学習方法を紹介します。

Web 上には Kotlin に関する素晴らしいブログやプレゼン資料が多くありますが、その資料がどのバージョンの Kotlin について言及しているのかにまず注意してください。正式リリース前は言語仕様の変更も多くあったので、古い情報は参考にならない場合もあります。

英語に抵抗のない方におすすめな方法は、公式ページ[*1]です。文法・ライブラリーなどの解説はもちろん、FAQ は一度目を通すことを強くおすすめします。

Kotlin の最新の情報を追いたいという場合は、Kotlin Weekly[*2]という週単位でのニュースや Kotlin の Podcast[*3]もおすすめです。こちらも英語です。

日本語で Kotlin について学びたい方におすすめなのは、書籍「Kotlin スタートブックー新しい Android プログラミングー（2016）長澤太郎著 リックテレコム」です。第 1 部の「初めての Kotlin」では概要が、第 2 部で「Kotlin 文法詳解」では言語機能・文法が、第 3 部「サンプルプログラミング」では Kotlin を使った Android 開発の例が紹介されています。自称 Kotlin エバンジェリストである著

[*1] https://kotlinlang.org/
[*2] http://www.kotlinweekly.net/
[*3] http://talkingkotlin.com/

者の豊富な Kotlin と Android 開発経験を元に書かれたこの書籍は、Android 開発者が Kotlin の学習の第一歩を踏み出す心強い見方になるでしょう。

実際に手を動かして学習したい方におすすめな方法は、Kotlin Koans online[*4] です。ブラウザーに提示されたコードに手を加えて、テストを通るようにするチュートリアルです。チュートリアルの説明は英語ですが、プログラミング文法に関する短い説明文ですので怖からずチャレンジしてみてください。

その他、Web 上には Kotlin に関する素晴らしいブログやプレゼン資料が多くあります。

「Kotlin 入門までの助走読本」としておすすめのブログをひとつ挙げるとすれば、藤原聖の「Google I/O 2017 : Introduction to Kotlin（和訳/要約）[*5]」です。Google I/O 2017 の Kotlin にまつわる内容を日本語で確認できる記事です。

「Kotlin 入門までの助走読本」の範囲は超えますが、より実践的で・詳しい内容が知りたいあなたは 2016 年の Kotlin アドベントカレンダー[*6]などもおすすめです。

繰り返しになりますが、記事や資料がどの Kotlin のバージョンについて言及しているか、まず注意してみてください。

3.2 コミュニティ

学習を進めるために、コミュニティとその仲間たちの存在は良いモチベーションになります。国内の Kotlin 界隈のコミュニティ事情はどのようになっているのでしょうか？

まず紹介するのは我々**日本 Kotlin ユーザグループ**（通称 JKUG）[*7]です。おそらく日本最古・最大人数の Kotlin コミュニティでしょう。また、世界の Kotlin ユーザグループ名簿[*8]に名を連ねています。Kotlin の発展や普及および知識共有を目的としており、勉強会の開催にとどまらず、日本語で Kotlin を語り合う

[*4] https://try.kotlinlang.org
[*5] http://qiita.com/satorufujiwara/items/490c1499acec1a921258
[*6] http://qiita.com/advent-calendar/2016/kotlin
[*7] https://kotlin.connpass.com/
[*8] http://kotlinlang.org/community/user-groups.html

Slack[*9]を管理したり、本書のような執筆プロジェクトを実施したりしています。

JKUG は現在のところ東京中心の活動ですが、**Kansai.kt**[*10]は関西を活動拠点とする Kotlin コミュニティです。

企業による Kotlin 勉強会の開催も多くあります。Sansan 株式会社による Kotlin 勉強会[*11]は、すでに 5 回開催されている比較的歴史の長い勉強会です。本書の筆者である Sansan エンジニアの山本と、ゲストスピーカーとして長澤は毎回登壇しており、参加者からライトニングトークを募集するスタイルの勉強会です。

株式会社サイバーエージェントの Kotlin 勉強会である「CA.kt[*12]」がスタートします。第 1 回が 2017 年 6 月 15 日に開催されます。執筆時（2017 年 5 月 25 日）現在で、定員 100 人に対し 186 人の参加希望者が集まっています。第 1 回はゲストスピーカーとして長澤が登壇し、サイバーエージェントのエンジニアとして本書執筆陣の中からは藤原と荒谷が登壇する予定です。勉強会以降に発表資料が公開されるでしょうから、残念ながら参加できない人も要チェックです。

世界中の Kotlin ファンと交流したいのならば、本家 Slack[*13]がおすすめです。また、今年 2017 年の 11 月 2 日と 3 日にサンフランシスコで **Kotlin Conf**[*14]という Kotlin カンファレンスが開催されます。

3.3 次回リリースの予告

今回のリリースが最初で最後というわけではありません。継続して（少なくとも、あと 1 回以上は）新バージョンをリリースしようと計画しています。というのは、今回スピード重視で執筆にあたりました。内容の正確さには自信を持っていますが、完全性や読みやすさという観点で粗が少なくないと思っています。次回以降で扱う範囲を増やし、構成や文章などにも磨きをかけていきたいです。

[*9] http://kotlinlang-jp.herokuapp.com
[*10] https://kansai-kt.connpass.com/
[*11] https://sansan.connpass.com/
[*12] https://cyberagent.connpass.com/event/57963/
[*13] http://slack.kotlinlang.org/
[*14] https://www.kotlinconf.com/

特に、今回のリリースでは含まれていない次のトピックを、次バージョンで解説する予定です。

- スコープ関数とその応用例
- キャスト
- デリゲーション

なお、本書はあくまで「入門までの助走」にフォーカスしています。あえて紹介しない Kotlin の文法や機能もあるでしょう。本章で示した「次のステップ」で Kotlin マスターへの道を歩んでください。

参考文献

[1] Kotlin native repository, GitHub

https://github.com/JetBrains/kotlin-native

[2] The Advent of Kotlin: A Conversation with JetBrains' Andrey Breslav', Oracle

http://www.oracle.com/technetwork/articles/java/breslav-1932170.html

[3] JVM Languages Report extended interview with Kotlin creator Andrey Breslav, ZEROTURNAROUND

https://zeroturnaround.com/rebellabs/jvm-languages-report-extended-interview-with-kotlin-creator-andrey-breslav/

[4] Compilation speed, Kotlin discuss

https://discuss.kotlinlang.org/t/compilation-speed/650/2

[5] Kotlin vs Java: Compilation speed, keepsafe

https://medium.com/keepsafe-engineering/kotlin-vs-java-compilation-speed-e6c174b39b5d

[6] Kotlin スタートブック 長澤太郎著 リックテレコム

著者一覧

本書の著者を紹介します（名前昇順）。自分たちで言うのもなんですが、粒揃いの精鋭です！

Abe Asami（きの子）

- Twitter: @aa7th
- ブログ: http://nocono.net/

大阪でフリーランスプログラマやってます。Scala 関西や関西 Java 女子部の主催もやってます。ゲームとマンガと音楽（ロック）が大好き。ビールが好きだけど、糖質を気にして最近 1 杯目は大体ハイボール。

荒谷（あらたに） 光（あきら）

- Twitter: @_a_akira
- ブログ: http://aakira.hatenablog.com

株式会社サイバーエージェントの Android エンジニア。デザインも好き。藤原さんと共に FRESH!というサービスの立ち上げ期から今も FRESH!を Kotlin で開発しているので、単純 Kotlin プレイ時間は多分この中でも長い方。毎日の為替グラフと毎週末の軟式テニスが楽しい。お酒は弱いのでビールは 1, 2 杯飲みます。しらふでも酔ってる人のテンションに合わせられるので問題ないです。

磯貝 佳典（いそがい よしのり）

- Twitter: @shiraj_i
- ブログ: http://shiraji.github.io

Kotlin コントリビュータ。株式会社アシックスでスマホアプリやサーバサイド開発に関わっています。ビールよりカクテル派。飲むと声が余計にでかくなります。

長澤 太郎（ながさわ たろう）

- Twitter: @ngsw_taro
- ブログ: http://taro.hatenablog.jp

自称 Kotlin エバンジェリストで日本 Kotlin ユーザグループ代表。エムスリー株式会社でソフトウェアエンジニアとして、世界の医療を変革するため日々励んでいます。静かなビーチで冷たいビールが飲みたい。

藤田 琢磨（ふじた たくま）

- Twitter: @magie_pooh

モバイルアプリ開発エキスパート養成読本で Kotlin の章を書いたりしてました。普段は株式会社 FOLIO で Android や Web フロントと戯れています。ビールは主に欧州（特にベルギービール）が専門。自宅には常時 20 種程のビールを用意しています。

藤原 聖（ふじわら さとる）

- Twitter: @satorufujiwara

株式会社サイバーエージェントの Android エンジニア。FRESH!というサービスの立ち上げを期に、2015 年から Kotlin で開発しています。Shibuya.apk とい

う Android 開発者コミュニティのオーガナイザーの 1 人。ビールも好きですが日本酒が好きです。

室星 亮太（むろほし りょうた）

- Twitter: @RyotaMurohoshi
- ブログ: http://mrstar-logs.hatenablog.com/

フラー株式会社所属プログラマ。Joren というサービスで Kotlin を導入。趣味はゲーム開発。Microsoft MVP for Visual Studio and Development Technologies 2016/10/01~。ビールも日本酒も好きなんですが最近お腹が。。。

八木 俊広（やぎ としひろ）

- Twitter: @sys1yagi
- ブログ: http://sys1yagi.hatenablog.com

WEB+DB PRESS Vol.94 で Kotlin 特集書いたりしました。普段は株式会社トクバイで Android エンジニア 兼 技術部長やってます。ノンアルコールビールはパフォーマンスが落ちないので便利。

山本 純平（やまもと じゅんぺい）

- Twitter: @boohbah

Sansan 株式会社で Eight というアプリの Android 版を作っています。Kotlin 勉強@Sansan を主催しています。最後までビールでいける派。

yy_yank（やんく）

- Twitter: @yy_yank
- ブログ: http://yyyank.blogspot.jp

Java プログラマです。Abby という会社で働いてます。たまに日本 Kotlin

ユーザーグループのお手伝いとかしてます。美味しかったお酒の名前を忘れるのが特技です。ビールは忘れにくいので好きです。

Kotlin 入門までの助走読本

2017 年 5 月 29 日　初版第 1 刷　発行
著　者　Abe Asami(きの子)、荒谷 光、磯貝 佳典、長澤 太郎、藤田 琢磨、藤原 聖、
印刷所　日本 Kotlin ユーザグループ