1 Kotlin. Что это за язык и зачем он нужен

В этой главе

- ✓ Демонстрация базовых возможностей Kotlin.
- ✓ Основные характеристики языка Kotlin.
- ✓ Возможности для серверной и Android-разработки.
- ✓ Обзор технологии Kotlin Multiplatform.
- ✓ Ключевые отличия Kotlin от других языков.
- ✓ Написание и запуск кода на Kotlin.

Kotlin — современный язык программирования на *виртуальной машине Java* (Java virtual machine, JVM) и не только. Это лаконичный, безопасный и прагматический язык программирования общего назначения. Его используют как независимые программисты, так и небольшие компании по разработке программного обеспечения и крупные предприятия. Множество разработчиков предпочитают Kotlin для написания мобильных приложений, создания приложений для серверов и персональных компьютеров (ПК), а также для достижения ряда других целей.

Kotlin задумывался как «улучшенная версия Java» — язык с расширенной эргономикой для разработчиков, позволившей исключить распространенные категории ошибок. Вдобавок в нем применяются современные парадигмы проектирования языка, но при этом сохранена возможность применения во всех задачах, где использовался язык Java. За последнее десятилетие Kotlin успел зарекомендовать себя

как прагматичный инструмент, подходящий для многих стилей разработки, типов проектов и платформ. Он стал преобладающим языком разработки для платформы Android. При создании программного обеспечения (ПО) на стороне сервера Kotlin стал мощной альтернативой Java. Для распространенных фреймворков, таких как Spring, или специализированных Kotlin-фреймворков, таких как Ktor, существует хорошая документация и поддержка.

В Kotlin сочетаются рабочие идеи существующих языков и инновационные подходы, такие как корутины для асинхронного программирования. Изначально язык был ориентирован только на JVM, но теперь существенно расширен: в нем есть поддержка кросс-платформенных решений. В этой главе мы подробно рассмотрим основные характеристики Kotlin.

1.1. ЗНАКОМСТВО С KOTLIN

Начнем с небольшого примера, чтобы продемонстрировать, как выглядит код на языке Kotlin (листинг 1.1). Даже в коротком ознакомительном фрагменте кода можно заметить множество интересных возможностей и концепций Kotlin — все они будут подробно обсуждаться в книге:

- определение класса данных `Person` со свойствами без необходимости указывать тело класса;
- объявление с помощью ключевого слова `val` свойств только для чтения (`name` и `age`);
- предоставление значений по умолчанию для аргументов;
- явная работа с нулевыми значениями (`Int?`) в системе типов, избегание так называемых ошибок на миллиард долларов — `NullPointerException`;
- определения функций верхнего уровня без необходимости оформлять вложенность в классы;
- именованные аргументы при вызове функций и конструкторов;
- использование висящих (или завершающих) запятых;
- использование операций над коллекциями с помощью лямбда-выражений;
- предоставление резервных значений с помощью оператора объединения с `null` (`?:`), когда переменная равна `null`;
- использование строковых шаблонов в качестве альтернативы выполнения конкатенации вручную;
- использование автоматически генерируемых функций (таких как `toString`) для классов данных.

К коду дано краткое пояснение, но если что-то пока будет вам непонятно, то не пугайтесь. В процессе изучения материала книги мы уделим достаточно времени описанию каждой детали этого листинга, и вы сможете писать подобный код самостоятельно.

Листинг 1.1. Знакомство с языком Kotlin

```
data class Person(   <-- Класс данных
    val name: String,   <-- Свойство только для чтения
    val age: Int? = null   <-- Тип (Int?), допускающий значение null, — значение по умолчанию для аргумента
)

fun main() {   <-- Функция верхнего уровня
    val persons = listOf(
        Person("Alice", age = 29),   <-- Именованный аргумент
        Person("Bob"),   <-- Висящая запятая
    )
    val oldest = persons.maxBy {   <-- Лямбда-выражение
        it.age ?: 0   <-- Оператор объединения с null
    }
    println("The oldest is: $oldest")   <-- Строковый шаблон
}

// The oldest is: Person(name=Alice, age=29)   <-- Автоматически сгенерированная функция toString
```

В этом фрагменте кода на языке Kotlin показано, как создать коллекцию, заполнить ее объектами класса Person, а затем найти самого старшего человека в коллекции, используя значения по умолчанию, если возраст не указан. При создании списка людей в нем не указан возраст записи с именем Bob, поэтому в качестве значения по умолчанию используется `null`. Для поиска самого старшего человека в списке используется функция `maxBy`. В функцию передается лямбда-выражение, которое принимает один параметр, по умолчанию названный `it` (хотя вы можете присвоить параметру и другие имена). Оператор объединения с `null` (`?:`) (Elvis operator) возвращает нулевое значение, если параметр `age` равен `null`. Возраст для записи Bob не указан, поэтому оператор `?:` заменяет данное значение нулем, и Alice считается записью, содержащей наибольшее значение возраста.

Попробуйте запустить пример самостоятельно. И самый простой способ сделать это — воспользоваться онлайн-площадкой на сайте https://play.kotlinlang.org/. Введите код примера и нажмите кнопку Run (Запуск), после чего код будет выполнен.

Вам понравился этот пример? Продолжайте читать книгу, изучайте язык и станьте экспертом в Kotlin. Надеемся, что вскоре такой код появится в ваших проектах, а не только на страницах нашей книги.

1.2. ОСНОВНЫЕ ХАРАКТЕРИСТИКИ KOTLIN

Kotlin — мультипарадигменный язык. Он статически типизирован, и это означает, что множество ошибок можно распознать во время компиляции, а не во время выполнения. В Kotlin сочетаются идеи объектно-ориентированных и функциональных языков, что помогает писать аккуратный код и использовать дополнительные мощные абстракции. Он предоставляет эффективный способ написания асинхронного кода, что важно во многих сферах разработки ПО.

Возможно, получив краткое описание, вы уже сформировали интуитивное представление о том, к какому типу языков относится Kotlin. Рассмотрим ключевые атрибуты языка более подробно. И для начала поговорим о том, какие типы приложений можно создавать с помощью Kotlin.

1.2.1. Варианты использования Kotlin: Android, серверная часть, везде, где работает Java, и не только

Диапазон целевых платформ Kotlin достаточно обширный. Язык не сосредоточен на какой-то одной проблемной области и не решает задачи какого-то определенного типа. Напротив, он позволяет повысить производительность для любых типов задач, возникающих в процессе разработки. При создании Kotlin в него была заложена идея высокого уровня интеграции с библиотеками, поддерживающими конкретные домены или парадигмы программирования.

В общем случае Kotlin применяется для создания:

- мобильных приложений, работающих на устройствах Android;
- кода на стороне сервера (как правило, бэкенд-веб-приложений).

Изначально Kotlin создавался в качестве более краткой, производительной и безопасной альтернативы Java, пригодной для использования во всех контекстах, в которых может использоваться Java. А это широкий спектр вариантов среды, от небольших периферийных устройств до крупнейших центров обработки данных. Во всех этих случаях Kotlin подходит идеально, и разработчики могут выполнять свою работу, имея дело с более кратким кодом и сталкиваясь с меньшим количеством неприятностей.

Однако Kotlin отлично работает и в других контекстах. С помощью технологии Kotlin Multiplatform можно создавать кросс-платформенные приложения для ПК, iOS и Android — и даже запускать программы на Kotlin в браузере. Эта книга преимущественно посвящена самому языку и тонкостям работы с JVM, а подробную информацию о других вариантах использования Kotlin можно найти на сайте https://kotl.in/. Далее рассмотрим ключевые качества Kotlin как языка программирования.

1.2.2. Статическая типизация повышает производительность, надежность и удобство сопровождения

У языков программирования со статической типизацией есть ряд преимуществ: производительность, надежность, удобство сопровождения и инструментальная поддержка. Ключевое же преимущество заключается в том, что тип каждого выражения в программе известен во время компиляции. Kotlin относится именно к таким языкам программирования. Его компилятор может подтвердить, что методы и поля, к которым вы пытаетесь получить доступ в объекте, действительно существуют. Это позволяет устранить целый класс ошибок — если поле отсутствует или тип возвращаемой функции не соответствует ожидаемому, то эти проблемы

будут известны в процессе компиляции, а не во время выполнения программы. Благодаря этому вы получаете возможность исправить проблемы на более ранних этапах разработки.

Перечислим некоторые преимущества статической типизации.

- *Производительность*. Вызов методов происходит быстрее, поскольку нет необходимости определять, какой метод нужно вызвать во время выполнения.
- *Надежность*. Компилятор использует типы для проверки согласованности программы, что снижает вероятность сбоев во время выполнения.
- *Удобство сопровождения*. Работать с незнакомым кодом проще, поскольку сразу понятно, с какими типами работает код.
- *Инструментальная поддержка*. Статическая типизация обеспечивает надежное выполнение рефакторинга, точное завершение кода и позволяет использовать иные возможности IDE.

Это серьезные отличия от *динамически типизированных* языков программирования, например Python или JavaScript. В этих языках существует возможность определять переменные и функции, которые могут хранить или возвращать данные любого типа, и разрешать ссылки на методы и поля во время выполнения. При таком подходе сокращается объем кода и повышается гибкость при создании структур данных. Но есть и обратная сторона: такие проблемы, как неправильно написанные имена или недопустимые параметры функций, невозможно обнаружить при компиляции, что, в свою очередь, может привести к ошибкам во время выполнения.

Тип каждого выражения в программе должен быть *известен* во время компиляции, однако при написании кода на Kotlin нет необходимости явно *указывать* тип каждой переменной. Как правило, тип переменной автоматически определяется из контекста, что позволяет обойтись без его объявления. Ниже приведен простейший пример этого механизма.

```
val x: Int = 1  ◄── Можно явно определить тип переменной
val y = 1  ◄── Но зачастую этого не требуется
```

При объявлении переменной она инициализируется целочисленным значением, и Kotlin автоматически определяет, что ее тип — `Int`. Способность компилятора определять типы из контекста называется *выведением типов* (не путать с приведением типов. — *Примеч. пер.*). В Kotlin этот механизм позволяет сократить объем кода, поскольку нет необходимости объявлять типы в явном виде.

К числу особенностей системы типов Kotlin можно отнести множество концепций, характерных для других объектно-ориентированных языков программирования. Например, поведение классов и интерфейсов ничем не отличается. А если у вас уже есть опыт в Java-разработке, то ваши знания особенно легко переносятся на Kotlin, и это касается в том числе таких тем, как обобщения.

В Kotlin есть поддержка *типов, допускающих значение null* (nullable types), позволяющая писать более надежные программы. Этот механизм призван обнаруживать

возможные исключения указателя `null` во время компиляции, чтобы сбоев такого рода не возникало во время выполнения. Мы вернемся к данной теме в подразделе 1.4.3 и подробно обсудим ее в главе 7, в которой также приводится сравнение с другими, возможно знакомыми вам, подходами к значениям `null`.

Кроме того, система типов Kotlin поддерживает *функции первого класса*. Чтобы разобраться, что это такое, рассмотрим основные идеи функционального программирования и определим, как эта концепция поддерживается в языке Kotlin.

1.2.3. Сочетание функционального и объектно-ориентированного программирования делает Kotlin безопасным и гибким

Kotlin — мультипарадигменный язык программирования, поэтому сочетает в себе *объектно-ориентированный подход* и *функциональный стиль программирования*. К ключевым концепциям функционального программирования относятся:

- *функции (фрагменты поведения) первого класса* — с ними можно работать как со значениями. Допускается хранить их в переменных, передавать в качестве параметров или возвращать из других функций;
- *неизменяемость* — работа выполняется с неизменяемыми объектами, что гарантирует их перманентное состояние после создания;
- *отсутствие побочных эффектов* — в коде будут написаны *чистые функции*, которые возвращают один и тот же результат при одинаковых входных данных, не изменяют состояние других объектов и не взаимодействуют с внешним миром.

Какие преимущества дает написание кода в функциональном стиле? Первое — *краткость* кода. Функциональный код может быть более аккуратным и лаконичным по сравнению с *императивным* аналогом: вместо изменения переменных и формирования логики в расчете на циклы и условные ветвления вы, работая с функциями как со значениями, получаете гораздо больше возможностей для абстрагирования.

К тому же применение такого стиля программирования позволит избежать дублирования в коде. Если несколько фрагментов кода со схожей логикой выполняют аналогичные задачи, то можно легко извлечь общую составляющую логики в отдельную функцию, а различающиеся части передавать в качестве аргументов. И эти аргументы, в свою очередь, тоже могут быть функциями. В Kotlin их можно выразить с помощью лаконичного синтаксиса лямбда-выражений.

Второе преимущество функционального кода — *безопасное конкурентное выполнение*. В многопоточных программах изменение одних и тех же данных несколькими «акторами» (часто несколькими потоками) без надлежащей синхронизации часто становится серьезным источником проблем. Использование неизменяемых структур данных и чистых функций позволяет гарантировать, что такие небезопасные изменения данных не произойдут. Следовательно, нет необходимости создавать сложные схемы синхронизации.

Наконец, функциональное программирование влечет за собой *более легкое тестирование*. Функции без побочных эффектов можно тестировать изолированно. В таком случае не требуется писать большое количество кода настройки, чтобы создать среду, от которой зависят эти функции. Когда область действия функций ограничена, вам легче рассуждать о своем коде и проверять его поведение, не учитывая постоянно особенности большой и сложной системы.

В целом функциональный стиль программирования можно использовать со многими языками, и большое количество аспектов этого подхода пропагандируется как хороший стиль программирования. Но не во всех языках реализована синтаксическая и библиотечная поддержка его использования. При создании в Kotlin закладывался обширный набор возможностей для поддержки функционального программирования:

- *типы функций* — возможность для функций принимать другие функции в качестве аргументов или возвращать другие функции;
- *лямбда-выражения* — позволяют передавать блоки кода с минимальным количеством шаблонов;
- *ссылки на члены без вызова* — возможность использовать функции в виде значений и, например, передавать их в качестве аргументов;
- *классы данных* — предоставление краткого синтаксиса создания классов для хранения неизменяемых данных;
- *API стандартной библиотеки* — большой набор инструментов в стандартной библиотеке для работы с объектами и коллекциями в функциональном стиле.

Во фрагменте кода ниже показана цепочка действий, выполняемых с входной последовательностью. Получив заданную последовательность сообщений, код находит «всех уникальных отправителей непустых непрочитанных сообщений, отсортированных по их именам»:

```
messages
    .filter { it.body.isNotBlank() && !it.isRead }
    .map(Message::sender)
    .distinct()
    .sortedBy(Sender::name)
```

Вы можете использовать функции `filter`, `map` и `sortedBy` из стандартной библиотеки. Язык Kotlin поддерживает лямбда-выражения и ссылки на члены без вызова (например, `Message::sender`), поэтому аргументы для этих функций очень лаконичны.

Таким образом, при написании кода на языке Kotlin вы можете сочетать объектно-ориентированный и функциональный подходы и использовать инструменты, наиболее подходящие для решения поставленной задачи. Это позволяет вам использовать все возможности функционального стиля программирования в Kotlin, а при необходимости работать с изменяемыми данными и писать функции с побочными эффектами. И конечно, работа с фреймворками на основе интерфейсов и иерархии классов будет несложной.

1.2.4. Конкурентный и асинхронный код становится естественным и структурированным при использовании корутин

При создании приложения для сервера, ПК или мобильного телефона избежать *конкурентности* — одновременного выполнения нескольких частей кода — практически невозможно. Пользовательские интерфейсы должны оставаться отзывчивыми, пока в фоновом режиме выполняются длительные вычисления. При взаимодействии с сервисами в Интернете приложения часто реализуют несколько запросов одновременно. Аналогичным образом серверные приложения должны продолжать обслуживать входящие запросы, даже если один из запросов занимает гораздо больше времени, чем обычно. Все эти приложения должны работать *конкурентно*, одновременно выполняя несколько задач.

Создать конкурентность позволяют множество подходов: потоки, обратные вызовы, фьючерсы, промисы, реактивные расширения и многое другое. В Kotlin подход к решению задач конкурентного и асинхронного программирования реализован на основе *приостанавливаемых вычислений*, называемых *корутинами*. С их помощью код может приостановить свое выполнение и возобновить работу в более поздний момент.

В примере ниже происходит определение функции `processUser`, выполняющей три сетевых вызова: `authenticate`, `loadUserData` и `loadImage`.

```
suspend fun processUser(credentials: Credentials) {
    val user = authenticate(credentials)
    val data = loadUserData(user)
    val profilePicture = loadImage(data.imageID)
    // ...
}

suspend fun authenticate(c: Credentials): User { /* ... */ }
suspend fun loadUserData(u: User): Data { /* ... */ }
suspend fun loadImage(id: Int): Image { /* ... */ }
```

Даже продолжительные операции...

...можно определить последовательно, сверху вниз...

...без блокировки приложения

Для этого используется ключевое слово suspend

Сетевой вызов может занять продолжительное время. При осуществлении каждого запроса выполнение функции `processUser` *приостанавливается* в ожидании результата. Однако поток, в котором выполняется этот код (и, соответственно, само приложение), *не блокируется*. В ожидании результата работы функции `processUser` могут выполняться другие задачи, например ответ на пользовательский ввод. (Подробности приостановки функций описаны в разделе 14.1.)

Невозможно написать этот код последовательно в императивной манере, один вызов за другим, без блокировки основных потоков. А при использовании обратных вызовов или реактивных потоков такая простая последовательная логика сильно усложняется.

В следующем примере два изображения загружаются одновременно с помощью функции `async` (ее вы изучите в подразделе 14.6.3). Затем ожидается завершение

загрузки, инициируемое функцией `await`, а в качестве результата возвращается комбинация изображений (например, одно, наложенное на другое):

```
suspend fun loadAndOverlay(first: String, second: String): Image =
    coroutineScope {
        val firstDeferred = async { loadImage(first) }
        val secondDeferred = async { loadImage(second) }
        combineImages(firstDeferred.await(), secondDeferred.await())
    }
```

Начало загрузки первого изображения в новой корутине

Когда оба изображения загружены, они накладываются друг на друга и результат возвращается

Начало загрузки второго изображения в еще одной корутине

Структурированная конкурентность, описанная в главе 15, поможет управлять временем жизни создаваемых корутин. В этом примере два процесса загрузки запускаются структурированным образом (из одной и той же *области видимости корутины*). Это гарантирует, что при сбое одной загрузки вторая будет автоматически отменена.

Кроме того, корутины относятся к очень легковесным абстракциям. Это означает, что вы можете запускать миллионы конкурентных заданий, при этом не теряя в производительности. Корутины Kotlin вкупе с абстракциями *холодных* и *горячих потоков* (описаны в главе 16) становятся мощным инструментом создания конкурентных приложений.

Вся третья часть книги посвящена изучению тонкостей и особенностей корутин, а также тому, как применять их для наилучшего решения поставленных задач.

1.2.5. Kotlin можно использовать для любых целей: он бесплатный и имеет открытый исходный код

Язык Kotlin, в том числе компилятор, библиотеки и все сопутствующие инструменты, полностью открыт и может свободно использоваться в любых целях. Он доступен под лицензией Apache 2: разработка ведется в открытом доступе на GitHub (http://github.com/jetbrains/kotlin).

Внести свой вклад в развитие Kotlin и его сообщества вы можете множеством разных способов.

- Приветствуется вклад в разработку новых функций и исправлений, связанных с компилятором Kotlin и сопутствующим инструментарием.
- Можно помочь улучшить опыт разработки на Kotlin, предоставляя сообщения об ошибках и отзывы.
- Потенциальные новые возможности языка подробно обсуждаются в сообществе, и вклад разработчиков на языке Kotlin играет большую роль в продвижении и развитии языка.

Кроме того, доступен выбор между несколькими IDE с открытым исходным кодом для разработки приложений на Kotlin: IntelliJ IDEA Community Edition и Android Studio, обе имеют полную поддержку. (Конечно, версия IntelliJ IDEA Ultimate тоже подходит для работы.)

Мы обсудили, что представляет собой язык Kotlin, а теперь посмотрим, как преимущества Kotlin влияют на решение конкретных практических задач.

1.3. ОБЛАСТИ, В КОТОРЫХ ЧАСТО ИСПОЛЬЗУЕТСЯ KOTLIN

Как мы уже говорили, две основные сферы использования Kotlin — серверная часть ПО и разработка под операционную систему (ОС) Android. Рассмотрим эти области применения и выясним, почему Kotlin хорошо подходит для них.

1.3.1. Обеспечение работы бэкенда — разработка серверной части ПО на Kotlin

Программирование на стороне сервера — довольно широкое понятие. В него входят перечисленные ниже типы приложений и не только:

- веб-приложения, возвращающие браузеру HTML-страницы;
- бэкенд для мобильных или одностраничных приложений, предоставляющих JSON API по протоколу HTTP;
- микросервисы, взаимодействующие с другими микросервисами по протоколу RPC или шине сообщений.

Разработчики уже много лет создают подобные приложения на базе JVM и накопили огромный стек фреймворков и технологий, с помощью которых их можно создавать. Такие приложения обычно не разрабатываются изолированно или с нуля. Почти всегда есть определенная система. Ее расширяют, улучшают или заменяют, и новый код должен интегрироваться с существующими частями системы, возможно написанными много лет назад.

В такой среде Kotlin особенно выигрывает за счет своей легкой совместимости с существующим кодом Java. Kotlin отлично подойдет как для переноса существующего сервиса на его кодовую базу, так и для написания нового компонента. У вас не возникнет проблем, когда понадобится расширить Java-классы в Kotlin или аннотировать методы и поля класса определенным образом. В то же время вы получите преимущества в виде компактности кода, его повышенной надежности и простоты сопровождения.

Еще одно большое преимущество Kotlin — повышенная надежность создаваемого приложения. Система типов Kotlin с точным отслеживанием значений `null` делает проблему исключений с `null`-указателями гораздо менее актуальной. Бóльшая часть кода на Java, вызывающего `NullPointerException` во время выполнения, в Kotlin

не компилируется. Поэтому исправить ошибку можно до того, как приложение попадет в среду эксплуатации.

Современные фреймворки, такие как Spring (https://spring.io/), обеспечивают поддержку функций первого класса в Kotlin. Помимо полной совместимости, эти фреймворки содержат дополнительные расширения и используют приемы, благодаря которым создается впечатление, что изначально они были разработаны для Kotlin.

В листинге 1.2 содержится простое приложение Spring Boot, передающее список объектов класса `Greeting`, стоящий из идентификатора и некоего текста в формате JSON, который передается по протоколу HTTP (рис. 1.1). Концепции из фреймворка Spring непосредственно переходят в Kotlin — используются те же аннотации (`@SpringBootApplication`, `@RestController`, `@GetMapping`), что и при работе с Java.

Листинг 1.2. Написание приложений Spring Boot на Kotlin

```
@SpringBootApplication
class DemoApplication

fun main(args: Array<String>) {
    runApplication<DemoApplication>(*args)
}

@RestController
class GreetingResource {
    @GetMapping
    fun index(): List<Greeting> = listOf(
        Greeting(1, "Hello!"),
        Greeting(2, "Bonjour!"),
        Greeting(3, "Guten Tag!"),
    )
}
data class Greeting(val id: Int, val text: String)
```

Функциональность фреймворка используется в виде аннотаций...

...при этом применяются выразительные возможности Kotlin

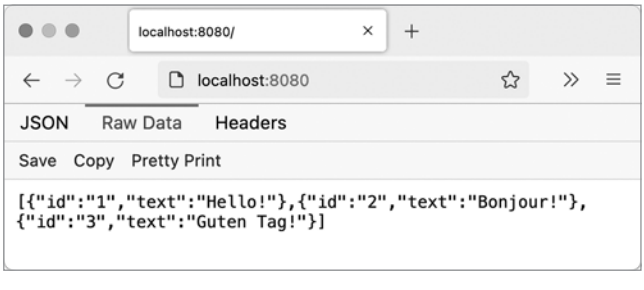

Рис. 1.1. Благодаря сочетанию Kotlin с такими проверенными в индустрии фреймворками, как Spring, создать приложение для предоставления JSON-файла по протоколу HTTP можно, написав всего несколько десятков строк кода

На сайтах Kotlin или Spring вы можете получить дополнительную информацию об использовании Spring совместно с Kotlin (https://kotlinlang.org/docs/jvm-get-started-spring-boot.html).