Оглавление

Содержание

Предисловие

Kotlin появился еще в 2011 году, но по-прежнему остается одним из самых новых языков в экосистеме Java. С тех пор появилась версия Kotlin, работающая на виртуальной машине JavaScript, а также версия, выполняющая компиляцию в машинный код. Это делает Kotlin гораздо более универсальным языком, чем Java, хотя между этими версиями есть большие различия, потому что версия для виртуальной машины Java опирается на стандартную библиотеку Java, недоступную в двух других версиях Kotlin. Сотрудники компании JetBrains, где создан язык Kotlin, прикладывают все силы, чтобы вывести все версии на один уровень, но, как вы понимаете, это требует времени.

Версия для JVM (Java Virtual Machine – виртуальной машины Java), безусловно, является наиболее широко используемой, и она получила дополнительный импульс к развитию, когда в Google объявили Kotlin одним из официальных языков разработки приложений для Android. Основная причина такого решения Google состоит в том, что последняя доступная в Android версия Java – это Java 6, тогда как Kotlin предлагает большинство особенностей Java 11 и многое другое. Kotlin также был объявлен официальным языком сценариев сборки в Gradle вместо Groovy, что позволяет использовать один и тот же язык и для сборки программ, и для самих собираемых программ.

В первую очередь язык Kotlin ориентирован на программистов на Java. Вполне возможно, что в будущем они будут изучать Kotlin как основной язык. Но пока основная масса программистов приходит в Kotlin из Java.

У каждого языка свой путь, определяемый некоторыми фундаментальными понятиями. Язык Java создавался на основе нескольких мощных идей. Он должен работать везде, т. е. в любой среде, где доступна JVM. Главный девиз: «Пиши один раз, запускай где угодно». Хотя некоторые могут утверждать обратное, но в целом Java соответствует этому девизу. Более того, теперь можно запускать почти везде не только программы на Java, но и программы на других языках, скомпилированные для JVM. Kotlin – один из таких языков.

Еще одна основополагающая идея Java – никакие нововведения никогда не станут причиной неработоспособности существующего кода. Хотя эта идея соблюдалась не всегда, но все же разработчики языка старались следовать ей. Конечно, это не всегда хорошо. Основным следствием является невозможность внесения в Java многих улучшений, имеющихся в других языках, потому что они могут привести к нарушению совместимости. Любая программа, скомпилированная с предыдущей версией Java, должна оставаться работоспособной в более новых версиях без перекомпиляции. Является ли это полезным или нет, это спорный вопрос, но в результате стремление максимально сохранить обратную совместимость постоянно играет против развития Java.

Также предполагалось, что Java сделает программы более безопасными за счет использования контролируемых исключений, что вынуждает программистов принимать эти исключения во внимание. Для многих это оказалось тяжким бременем, ведущим к практике постоянного преобразования контролируемых исключений в неконтролируемые.

Хотя Java является объектно-ориентированным языком, он должен быть настолько же быстрым, как и большинство языков, используемых для вычислительных задач. В свое время разработчики языка решили, что Java только выиграет, если, помимо объектов, представляющих числа и логические значения, в языке будут поддерживаться элементарные числовые типы, позволяющие выполнять вычисления намного быстрее. Из-за этого отсутствует возможность помещать значения примитивных типов в коллекции, такие как списки, множества и ассоциативные массивы. А когда были добавлены потоки данных (streams), разработчики языка решили создать специальные версии для примитивов, но не для всех, а только для наиболее часто используемых. Если вы пользуетесь некоторыми из неподдерживаемых примитивов, значит, вам не повезло.

То же самое произошло с функциями. В Java 8 появилась поддержка обобщенных функций, но обобщение возможно только для объектов. Поэтому для обработки целых, длинных целых, вещественных чисел двойной точности и логических значений были разработаны специализированные функции. (И снова, к сожалению, поддержка добавлена не для всех элементарных типов – для работы с однобайтовыми и короткими целыми, а также для вещественных чисел одинарной точности нет специализированных функций.) Что еще хуже, потребовались дополнительные функции для преобразования из одного элементарного типа в другой или из элементарных типов в объекты и обратно.

Язык Java был разработан более 20 лет назад. С тех пор многое изменилось, но большинство этих изменений нельзя было перенести в Java, потому что это нарушило бы совместимость. Некоторые изменения все же вносились, и совместимость сохранилась, но за счет удобства использования.

Для устранения этих ограничений было создано много новых языков, таких как Groovy, Scala и Clojure. Они в определенной степени совместимы с Java и позволяют применять существующие библиотеки Java,

а программисты на Java могут использовать библиотеки, написанные на этих языках.

Kotlin пошел другим путем. Он гораздо сильнее интегрирован с Java, что позволяет без проблем смешивать исходный код на Kotlin и Java в одном проекте! В отличие от других языков для JVM Kotlin не выглядит сильно отличающимся от Java (хотя разница, конечно же, имеется). Он больше похож на язык, которым должен стать язык Java. Некоторые даже говорят, что Kotlin – это Java, созданный правильно, потому что решает большинство проблем, характерных для Java. (Однако Kotlin пока не предлагает решений, связанных с ограничениями, свойственными JVM.)

Но, что еще более важно, Kotlin разрабатывался так, чтобы быть намного более восприимчивым ко многим новым приемам, появляющимся в функциональном программировании. В Kotlin есть изменяемые и неизменяемые ссылки, но предпочтение отдается неизменяемым. Kotlin также поддерживает большую часть абстракций функционального программирования, которые позволяют избегать управляющих структур (хотя он имеет традиционные управляющие структуры, помогающие упростить переход от традиционных языков).

Еще одна замечательная особенность Kotlin – он уменьшает потребность в шаблонном коде, позволяя сократить его до минимума. На Kotlin можно написать класс с необязательными свойствами (обладающий также функциями `equals`, `hashCode`, `toString` и `copy`) в одной строке кода, тогда как для объявления эквивалентного класса на Java потребуется около тридцати строк (включая методы свойств и перегруженные конструкторы).

Да, есть другие языки программирования, разработанные для преодоления ограничений Java в среде JVM, но Kotlin отличается тем, что прекрасно интегрируется с Java-программами на уровне исходного кода. Вы можете смешивать исходные файлы на Java и Kotlin в проектах и использовать единую цепочку сборки. Это меняет правила игры, особенно в отношении командного программирования, потому что использование Kotlin в среде Java – не более сложная задача, чем использование любой сторонней библиотеки. Это обеспечивает максимально плавный переход с Java на новый язык и позволяет писать программы, которые:

- безопаснее;
- проще в разработке, тестировании и сопровождении;
- более масштабируемые.

Я полагаю, что многие читатели – программисты на Java – рано или поздно осознают необходимость поиска новых решений своих повседневных проблем. Возможно, у вас уже возник вопрос, почему вы должны использовать Kotlin. Нет ли других языков в экосистеме Java, которые позволят легко применять безопасные методы программирования?

Конечно, есть, и одним из самых известных является Scala. Scala – очень хорошая альтернатива Java, но у Kotlin есть нечто большее. Scala может

взаимодействовать с Java на уровне библиотек, т. е. программы на Java могут использовать библиотеки Scala (объекты и функции), а программы на Scala могут использовать библиотеки Java (объекты и методы). Но программы на Scala и Java должны создаваться как отдельные проекты или хотя бы как отдельные модули, тогда как классы Kotlin и Java можно смешивать внутри одного модуля.

Прочитайте эту книгу, чтобы узнать больше о Kotlin.

Благодарности

Я хотел бы поблагодарить всех, кто участвовал в создании этой книги. Прежде всего, большое спасибо моему редактору Марине Майклз (Marina Michaels). Мне было очень приятно работать с Вами. Также спасибо моему научному редактору Александару Драгосавлевичу (Aleksandar Dragosavljević).

Большое спасибо также Джоэлю Котарски (Joel Kotarski), Джошуа Уайту (Joshua White) и Риккардо Терреллу (Riccardo Terrell) – техническим редакторам, Алессандро Кампейсу (Alessandro Campeis) и Бренту Уотсону (Brent Watson) – корректорам, которые помогли мне сделать эту книгу намного лучше. Спасибо всем рецензентам, читателям, участвующим в программе MEAP, и всем, кто прислал свои отзывы и комментарии, спасибо вам! Эта книга не получилась бы такой, какая она есть, без вашей помощи. Также я хотел бы поблагодарить людей, которые нашли время, чтобы просмотреть и прокомментировать книгу: Алексея Слайковского (Aleksei Slaikovskii), Алессандро Кампейса (Alessandro Campeis), Энди Кирша (Andy Kirsch), Бенджамина Голдберга (Benjamin Goldberg), Бриджера Хауэлла (Bridger Howell), Конора Редмонда (Conor Redmond), Дилана Макнейми (Dylan McNamee), Эммануэль Мединy Лопес (Emmanuel Medina López), Фабио Фальси Родригес (Fabio Falci Rodrigues), Федерико Кирхейса (Federico Kircheis), Герго Михай Надя (Gergő Mihály Nagy), Грегора Раймана (Gregor Raýman), Джейсона Ли (Jason Lee), Жан-Франсуа Морина (Jean-François Morin), Кента Р. Спилнера (Kent R. Spillner), Линн Нортроп (Leanne Northrop), Марка Элстона (Mark Elston), Мэтью Халверсона (Matthew Halverson), Мэтью Проктора (Matthew Proctor), Нуно Александра (Nuno Alexandre), Рафаэля Вентальо (Raffaella Ventaglio), Рональда Харинга (Ronald Haring), Шило Морриса (Shiloh Morri), Винсента Терона (Vincent Theron) и Уильяма Э. Уилера (William E. Wheeler).

Хочу также поблагодарить сотрудников издательства Manning: Дейрдре Хиам (Deirdre Hiam), Фрэнсиса Бурана (Frances Buran), Кери Хейлза (Keri Hales), Дэвида Новака (David Novak), Мелоди Долаб (Melody Dolab) и Николь Берд (Nichole Beard).

О книге

Кому адресована эта книга

Цель этой книги – не просто помочь выучить язык Kotlin, но также научить писать гораздо более безопасные программы. Это не означает, что Kotlin следует использовать, только если вы решите писать более безопасные программы, и тем более не означает, что писать более безопасные программы можно только на Kotlin. Эта книга представляет примеры, написанные на Kotlin, потому что это один из самых дружественных языков для разработки безопасных программ в экосистеме JVM.

В книге рассказывается о методах, разработанных давно и в самых разных окружениях, хотя многие из них своими корнями уходят в функциональное программирование. Но эта книга не о фундаменталистском функциональном программировании, она о прагматичном безопасном программировании.

Все описанные здесь приемы годами использовались в экосистеме Java и доказали свою эффективность в разработке программ с гораздо меньшим количеством ошибок реализации, чем традиционные методы императивного программирования. Эти безопасные приемы можно реализовать на любом языке, и некоторые из них уже многие годы используются в Java, но часто для этого приходится преодолевать ограничения Java.

Эта книга не учит программированию с самого начала. Она писалась для профессиональных программистов, ищущих более простые и безопасные способы разработки безошибочных программ.

О чем вы узнаете

В этой книге вы познакомитесь с конкретными приемами, которые могут отличаться от уже знакомых вам, если прежде вы программировали на Java. Большинство из них покажутся вам незнакомыми или даже противоречащими приемам, которые программисты привыкли считать

оптимальными. Но многие (хотя и не все) оптимальные приемы были выработаны во времена, когда компьютеры имели 640 Кб памяти, 5 Мб дискового пространства и одноядерный процессор. Времена изменились. Сейчас простой смартфон – это настоящий компьютер с 3 Гб или более оперативной памяти, с твердотельным накопителем на 256 Гб и 8-ядерным процессором. Компьютеры тоже имеют много гигабайт оперативной памяти, терабайты дискового пространства и многоядерные процессоры.

В этой книге я расскажу о:

- дальнейшем увеличении абстракции;
- предпочтительном использовании неизменяемых объектов;
- ссылочной прозрачности;
- инкапсуляции общего изменяемого состояния;
- абстрагировании условных и управляющих структур;
- использовании правильных типов данных;
- использовании отложенных вычислений;
- и многом другом.

Дальнейшее увеличение абстракции

Один из наиболее важных методов, с которыми вы познакомитесь, – это дальнейшее увеличение абстракции (традиционные программисты привыкли считать преждевременную абстракцию злом, как и преждевременную оптимизацию). Но дальнейшее увеличение абстракции помогает лучше понять решаемую задачу, что в свою очередь гарантирует правильность решения.

Вы можете спросить, что в действительности подразумевается под дальнейшим увеличением абстракции. Здесь все просто – это умение распознавать типичные шаблоны в различных вычислениях и их абстрагирование, чтобы избежать повторяющегося кода.

Неизменяемость

Суть неизменяемости заключается в использовании только неизменяемых данных. Многим традиционным программистам трудно представить, как можно писать полезные программы, используя только неизменные данные. Разве программирование не основано в первую очередь на изменении данных? Если следовать такой логике, тогда бухгалтерский учет – это прежде всего изменение значений в бухгалтерской книге.

Переход от использования изменяемого к использованию неизменяемого учета произошел еще в XV веке, и с тех пор принцип неизменяемости был признан основным элементом безопасности для учета. Этот принцип также применим к программированию, как вы увидите в этой книге.

Ссылочная прозрачность

Ссылочная прозрачность позволяет писать детерминированные программы, т. е. программы, результаты работы которых можно предсказать

и осмыслить. Такие программы всегда дают одинаковые результаты для одних и тех же входных данных. Это не означает, что они всегда дают одинаковые результаты, но различия в результатах зависят только от изменений на входе, а не от внешних условий.

Такие программы не только безопаснее (потому что обладают предсказуемым поведением), но их также проще писать, сопровождать, обновлять и тестировать. А программы, которые легче тестировать, обычно проверяются более полно и, следовательно, получаются более надежными.

Инкапсуляции общего изменяемого состояния

Неизменяемые данные автоматически оказываются защищены от случайного изменения при совместном использовании, что часто вызывает много проблем при конкурентной и параллельной обработке данных, таких как взаимоблокировка, простои потоков выполнения и устаревание данных. Но отсутствие общего изменяемого состояния превращается в проблему, когда оно действительно необходимо. Это в первую очередь относится к конкурентному и параллельному программированию.

Устранением изменяемого состояния исключается возможность случайного обмена ошибочными данными, благодаря чему программы становятся безопаснее. Но конкурентное и параллельное программирование подразумевает совместное изменение общего состояния. Без этого конкурирующие и параллельные потоки выполнения не смогут сотрудничать друг с другом. Этот конкретный вариант использования изменяемого общего состояния можно абстрагировать и инкапсулировать так, чтобы получить возможность повторного использования без риска, потому что будет иметься одна полностью протестированная универсальная реализация.

В этой книге вы узнаете, как абстрагировать и инкапсулировать общее изменяемое состояние, чтобы реализовать его только один раз и потом использовать везде, где оно необходимо.

Абстрагирование условных и управляющих структур

Вторым распространенным источником ошибок в программах после общего изменяемого состояния являются управляющие структуры. Традиционные программы состоят из таких управляющих структур, как циклы и проверки условий. С этими структурами так легко ошибиться, что разработчики языка постарались максимально абстрагировать детали. Одним из лучших примеров является цикл `for-each`, который ныне присутствует в большинстве языков (хотя в Java он все еще называется `for`).

Другая распространенная проблема – правильное использование `while` и `do while` (или `repeat until`) и, в частности, выбор места, где должно проверяться условие. Еще одна проблема – конкурентное изменение элементов коллекций во время их обхода в цикле, когда проблема общего изменяемого состояния возникает даже при использовании единственного потока выполнения! Абстрагирование управляющих структур позволяет полностью устранить подобные проблемы.

Использование правильных типов данных

В традиционном программировании такие универсальные типы, как `int` и `String`, используются для представления величин без учета единиц измерения. Как следствие, очень легко допустить ошибку, сложив мили с галлонами или доллары с минутами. Использование типов значений может полностью устранить проблемы такого рода при очень низких затратах, даже если используемый язык не предлагает истинных типов значений.

Отложенные вычисления

Большинство распространенных языков называют *строгими*, в том смысле, что аргументы, передаваемые методу или функции, вычисляются заранее, перед их обработкой. На первый взгляд, такое поведение вполне оправданно, хотя часто это не так. Отложенные, или «ленивые», вычисления – это прием, заключающийся в откладывании вычисления элементов до момента, когда они действительно становятся необходимы. Программирование почти целиком основано на отложенных вычислениях.

Например, условие в конструкции `if...else` вычисляется немедленно, перед его проверкой, но выполнение ветвей откладывается, в том смысле, что выполняется только ветвь, соответствующая условию. Эта отложенность скрыта, и программист не контролирует ее. Явное использование отложенных вычислений помогает писать гораздо более эффективные программы.

Читатели

Эта книга адресована читателям, уже имеющим опыт программирования на Java. Необходимо также некоторое понимание параметризованных (обобщенных) типов. В этой книге широко используются параметризованные функции, или варианты, которые редко применяются в Java (хотя это мощный инструмент). Не пугайтесь, если вы еще не знакомы с этими приемами; я расскажу о них и объясню, зачем они нужны, когда подойдет время.

Структура книги

Книга предполагает последовательное чтение, потому что каждая следующая глава основана на понятиях, рассматриваемых в предыдущих главах. Я использую слово «чтение», но эта книга предназначена не только для чтения. Очень немногие разделы содержат одну лишь теорию.

Чтобы извлечь максимум пользы из книги, выполняйте все упражнения, встречающиеся в процессе чтения. Каждая глава содержит ряд упражнений с необходимыми инструкциями и подсказками, которые помогут вам прийти к решению. Каждое упражнение сопровождается предлагаемым решением и тестом, который вы можете использовать для проверки своего решения.