2. Переменные, константы и типы

В этой главе мы познакомим вас с переменными, константами и базовыми типами данных в Kotlin — фундаментальными элементами любой программы. *Переменные* и *константы* используются для хранения значений или передачи данных внутри приложения. *Типы* описывают конкретные данные, хранимые переменной или константой.

Есть важные различия между типами данных, а также между переменными и константами, которые и определяют порядок их использования.

Типы

Данные, хранимые в переменных и константах, относятся к определенному типу. Тип описывает данные, хранящиеся в константе или переменной, и указывает, как при компиляции будет происходить *проверка типа*. Такая проверка предотвращает присваивание переменной или константе данных неправильного типа.

Чтобы увидеть, как работает проверка типа, отредактируйте файл Main.kt из проекта bounty-board, созданный в главе 1. Если вы закрыли среду IntelliJ, запустите ее заново. Скорее всего, проект bounty-board откроется автоматически, так как IntelliJ при запуске открывает последний проект, с которым вы работали. Если этого не произошло, выберите bounty-board в списке недавних проектов в середине окна приветствия или при помощи команды File ▸ Open Recent ▸ bounty-board.

Объявление переменной

Представьте, что вы пишете игру-приключение, в которой игроку каждый раз предлагается выполнить некую миссию. Сложность миссии возрастает по мере того, как игрок становится сильнее и поднимается на следующий уровень в игре. Вероятно, вам понадобится переменная для хранения текущего уровня игрока.

В файле Main.kt создайте первую переменную с именем `playerLevel` и присвойте ей значение.

Листинг 2.1. Объявление переменной playerLevel (Main.kt)

```
fun main() {
    println("Hello, world!")
    var playerLevel: Int = 4
```

```
    println(playerLevel)
}
```

Здесь экземпляр типа `Int` присваивается переменной с именем `playerLevel`. Давайте подробнее рассмотрим, что у нас получилось.

Для определения переменной используется ключевое слово `var`, которое объявляет новую переменную. После ключевого слова мы задаем имя переменной.

Затем указывается тип переменной `Int`. Это означает, что в `playerLevel` будет храниться целое число.

И наконец, мы используем *оператор присваивания* (=), чтобы присвоить значение в правой части (экземпляр типа `Int`, а именно `4`) переменной в левой части (`playerLevel`).

На рис. 2.1 показано объявление `playerLevel` в виде диаграммы.

После объявления значение переменной можно вывести в консоль с помощью функции `println`.

Запустите программу, щелкнув на кнопке запуска рядом с функцией `main` и выбрав Run 'MainKt'. Также для этого можно воспользоваться кнопкой запуска на панели инструментов IntelliJ. В консоли отобразится число 4 — то значение, которое было присвоено `playerLevel`.

Рис. 2.1. Порядок объявления переменной

Теперь попробуйте присвоить `playerLevel` значение `"thirty-two"`. (Зачеркнутая строка означает, что код надо удалить.)

Листинг 2.2. Присваивание "thirty-two" переменной playerLevel (Main.kt)

```
fun main() {
    println("Hello, world!")
    var playerLevel: Int = 4
    var playerLevel: Int = "thirty-two"
    println(playerLevel)
}
```

Снова запустите `main` при помощи кнопки запуска. На этот раз компилятор Kotlin сообщит об ошибке:

```
e: Main.kt: (3, 28): Type mismatch: inferred type is String but Int was
                                    expected
```

Набирая код, вы могли заметить, что `"thirty-two"` подчеркнуто красным. Так IntelliJ показывает вам обнаруженную ошибку. Наведите указатель мыши на `"thirty-two"`, отобразится описание проблемы (рис. 2.2).

Рис. 2.2. Подсказка: обнаружено несоответствие типа

Kotlin использует *статическую типизацию*, то есть компилятор проверяет все типы в исходном коде, чтобы убедиться, что написанный код корректен. Также статическая типизация означает, что после определения переменной вы не сможете изменить тот тип, с которым она была объявлена.

IntelliJ проверяет код в процессе набора и обнаруживает ошибки, связанные с попытками присвоить переменной значение неверного типа. Такая возможность, называемая *статической проверкой согласованности типов*, помогает выявлять ошибки программирования еще до компиляции кода.

Чтобы устранить ошибку, надо присвоить переменной `playerLevel` другое значение типа `Int` — например, заменить `"thirty-two"` целым числом 4.

Листинг 2.3. Исправление ошибки типа (Main.kt)

```
fun main() {
    println("Hello, world!")
    var playerLevel: Int = "thirty-two"
    var playerLevel: Int = 4
    println(playerLevel)
}
```

Если вы соблюдаете правила назначения типов, в процессе выполнения переменной можно присвоить другое значение. Например, при повышении уровня игрока переменной `playerLevel` может быть присвоено новое значение. Например, `playerLevel` можно присвоить значение 5.

Листинг 2.4. Переменной playerLevel присваивается 5 (Main.kt)

```
fun main() {
    println("Hello, world!")
    var playerLevel: Int = 4
    println(playerLevel)

    println("The hero embarks on her[1] journey to locate the enchanted sword.")
```

[1] В примерах кода авторы используют местоимения женского рода she, her для героя игры (возможно, из соображений политкорректности). На русском языке «герой» и «игрок»

```
    playerLevel = 5
    println(playerLevel)
}
```

Снова выполните обновленную функцию `main`, чтобы увидеть, как работает присваивание. В консоли выводится число 5 в отдельной строке после сообщения `The hero embarks on her journey to locate the enchanted sword` (Герой отправляется в путешествие, чтобы найти заколдованный меч).

Оператор присваивания (`=`) связывает переменную с новым значением. Такое решение работает, но правильнее было бы увеличить `playerLevel` на `1`, вместо того чтобы присваивать переменной значение `5`. Увеличение уровня работает лучше, потому что ваш код станет более гибким — например, если вам понадобится изменить начальный уровень игрока.

Обновите операцию присваивания, чтобы значение переменной увеличивалось на `1`. Заодно обновите приветственное сообщение, которое сейчас кажется немного неуместным.

Листинг 2.5. Увеличение playerLevel (Main.kt)

```
fun main() {
    println("Hello, world!")
    println("The hero announces her presence to the world.")
    var playerLevel: Int = 4
    println(playerLevel)

    println("The hero embarks on her journey to locate the enchanted sword.")
    playerLevel = 5
    playerLevel += 1
    println(playerLevel)
}
```

После присваивания переменной `playerLevel` значения `4` используем *оператор сложения с присваиванием* (`+=`) для увеличения исходного значения на `1`. Запустите программу снова. Вы увидите новое приветствие, а уровень игрока изменится с 4-го на 5-й, как и прежде.

Kotlin также предоставляет другие возможности присваивания значений. Так как `playerLevel` увеличивается на `1`, вместо оператора `+=` можно использовать *оператор инкремента* (`++`):

```
playerLevel++
```

Для уменьшения значения на `1` можно использовать оператор декремента (`--`), а для уменьшения на произвольную величину — оператор вычитания с присваиванием (`-=`). Также существуют аналогичные операторы для умножения с при-

звучит более привычно, чем «героиня» или «игрокиня», поэтому в русской версии герой Мадригал будет мужского рода. — *Примеч. ред.*

сваиванием (`*=`) и деления с присваиванием (`/=`). Математические операторы мы рассмотрим более подробно в главе 5.

Встроенные типы языка Kotlin

Вы уже видели переменные типа `Int`, а также пользовались типом `String` при вызове функции `println`. Kotlin также поддерживает типы для работы со значениями «истина/ложь», списками и парами «ключ — значение». В табл. 2.1 перечислены наиболее часто используемые типы, доступные в Kotlin.

Таблица 2.1. Наиболее часто применяемые встроенные типы

Тип	Описание	Примеры
`String` (строка)	Текстовая информация	`"Madrigal"` `"happy meal"`
`Char` (символ)	Один символ	`'X'` Символ Юникод U+0041
`Boolean` (логический)	Истина/ложь Да/Нет	`true` `false`
`Int` (целочисленный)	Целое число	`5` `"Madrigal".length`
`Double` (с плавающей запятой)	Дробные числа	`3.14` `2.718`
`List` (список)	Коллекция элементов	`3, 1, 2, 4, 3` `"root beer", "club soda", "coke"`
`Set` (множество)	Коллекция уникальных значений	`"Larry", "Moe", "Curly", "Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune"`
`Map` (ассоциативный массив)	Коллекция пар «ключ — значение»	`"small" to 5.99, "medium" to 7.99, "large" to 10.99`

Если какие-то типы вам неизвестны, не переживайте — вы познакомитесь с ними в процессе чтения книги. В частности, строки рассматриваются в главе 6, числа — в главе 5, а списки, множества, ассоциативные массивы (относящиеся к категории *коллекций*) — в главах 9 и 10.

Переменные, доступные только для чтения

До настоящего времени вам попадались только переменные, которым можно присвоить новые значения; такие переменные называются *изменяемыми*. Но часто

возникает необходимость использовать переменные, значение которых должно оставаться постоянным все время выполнения программы. Например, в текстовой приключенческой игре имя игрока не должно меняться после начального присваивания.

Язык Kotlin позволяет объявлять переменные, *доступные только для чтения*, — значения таких переменных нельзя изменить после присваивания.

Переменная, которую можно изменить, объявляется с помощью ключевого слова `var`. Чтобы объявить переменную, доступную только для чтения, используется ключевое слово `val`.

Переменные, которые могут изменяться, мы будем называть `vars`, а переменные, доступные только для чтения, — `vals`.

Добавьте определение `val` для хранения имени игрока и выведите его после вывода начального уровня игрока.

Листинг 2.6. Добавление val heroName (Main.kt)

```
fun main() {
    println("The hero announces her presence to the world.")

    val heroName: String = "Madrigal"
    println(heroName)
    var playerLevel: Int = 4
    println(playerLevel)

    println("The hero embarks on her journey to locate the enchanted sword.")
    playerLevel += 1
    println(playerLevel)
}
```

Запустите программу при помощи кнопки запуска рядом с функцией `main` или в строке меню. Значения `playerLevel` и `heroName` должны появиться в консоли:

```
The hero announces her presence to the world.
Madrigal
4
The hero embarks on her journey to locate the enchanted sword.
5
```

Далее попытаемся присвоить `heroName` другое строковое значение оператором `=` и снова запустим программу.

Листинг 2.7. Попытка изменения значения heroName (Main.kt)

```
fun main() {
    println("The hero announces her presence to the world.")

    val heroName: String = "Madrigal"
    println(heroName)
```

```
    var playerLevel: Int = 4
    println(playerLevel)

    heroName = "Estragon"

    println("The hero embarks on her journey to locate the enchanted sword.")
    playerLevel += 1
    println(playerLevel)
}
```

При попытке запуска в консоли появится следующее сообщение об ошибке компиляции:

```
e: Main.kt: (9, 5): Val cannot be reassigned
```

Компилятор сообщил о попытке изменить `val`. После начального присваивания значение `val` нельзя изменить.

Удалите вторую операцию присваивания, чтобы исправить ошибку повторного присваивания.

Листинг 2.8. Исправление ошибки повторного присваивания значения val (Main.kt)

```
fun main() {
    println("The hero announces her presence to the world.")

    val heroName: String = "Madrigal"
    println(heroName)
    var playerLevel: Int = 4
    println(playerLevel)

    heroName = "Estragon"

    println("The hero embarks on her journey to locate the enchanted sword.")
    playerLevel += 1
    println(playerLevel)
}
```

`vals` полезны для защиты от случайного изменения значений переменных, которые предназначены только для чтения. По этой причине мы рекомендуем использовать `val` везде, где не требуется `var`.

Среда IntelliJ способна определить на основании статического анализа кода, когда `var` можно превратить в `val`. Если `var` не изменяется по ходу программы, IntelliJ предложит преобразовать его в `val`. Мы советуем следовать рекомендациям IntelliJ, если, конечно, вы не планируете писать код для изменения значений `var`. Чтобы увидеть, как выглядит рекомендация от IntelliJ, преобразуйте `heroName` в `var`.

Листинг 2.9. Замена heroName на var (Main.kt)

```
fun main() {
    println("The hero announces her presence to the world.")

    val heroName: String = "Madrigal"
    var heroName: String = "Madrigal"
    println(heroName)
    var playerLevel: Int = 4
    println(playerLevel)

    println("The hero embarks on her journey to locate the enchanted sword.")
    playerLevel += 1
    println(playerLevel)
}
```

Так как значение `heroName` нигде не изменяется, нет необходимости (и не следует) объявлять его как `var`. Обратите внимание, что среда IntelliJ выделила строку с ключевым словом `var` горчичным цветом. Если навести указатель мыши на ключевое слово `var`, IntelliJ сообщит о предлагаемом изменении (рис. 2.3).

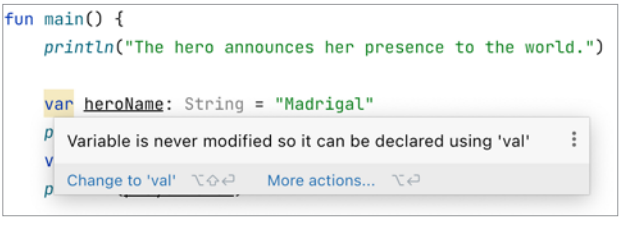

Рис. 2.3. Подсказка: переменная нигде не изменяется

Как и ожидалось, IntelliJ предлагает преобразовать `heroName` в `val`. Чтобы подтвердить изменение, щелкните на ключевом слове `var` рядом с `heroName` и нажмите Option-Return (Alt-Enter). В появившемся меню выберите Change to val (рис. 2.4).

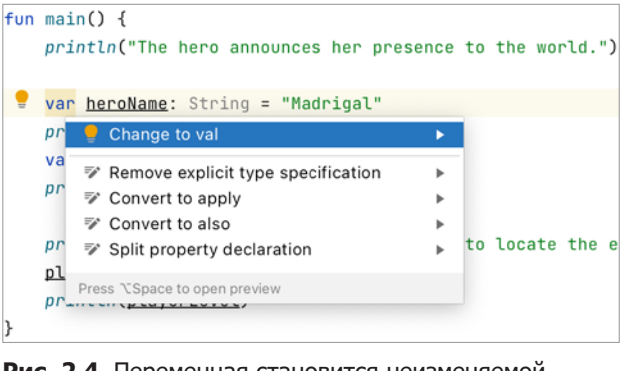

Рис. 2.4. Переменная становится неизменяемой

IntelliJ автоматически заменит `var` на `val`:

```
val heroName: String = "Madrigal"
println(heroName)
```

Как мы уже говорили, рекомендуется использовать `val` всегда, когда это возможно, чтобы компилятор Kotlin мог предупредить о случайных попытках присвоить ей другое значение. Также советуем не игнорировать предложения IntelliJ касательно возможного улучшения кода. Им можно и не следовать, но обратить внимание определенно стоит.

Автоматическое определение типов

Обратите внимание, что типы, которые вы указали для переменных `heroName` и `playerLevel`, выделены серым цветом в IntelliJ. Серый цвет показывает элементы, которые являются необязательными или не используются. Наведите указатель мыши на определение типа `String`, и IntelliJ объяснит, почему эти элементы необязательны (рис. 2.5).

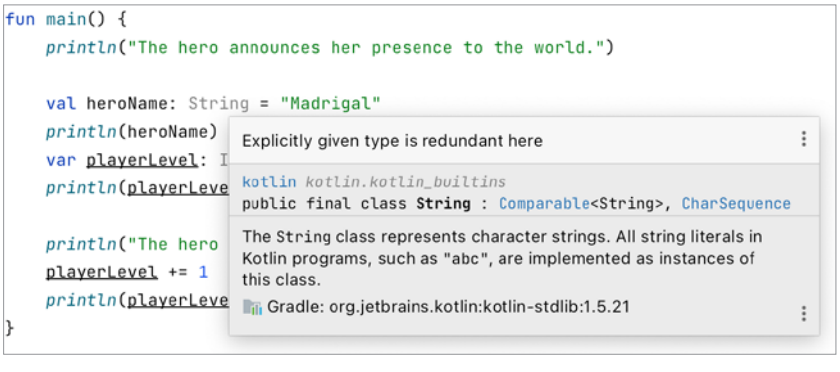

Рис. 2.5. Избыточная информация о типе

Как видите, Kotlin считает, что ваше объявление типа избыточно. Что это значит?

Kotlin поддерживает механизм *автоматического определения* типов (type inference), что позволяет опустить типы для переменных, которым присваиваются значения при объявлении. Так как при объявлении переменной `heroName` присваивается значение типа `String` и переменной `playerLevel` присваивается значение типа `Int`, компилятор Kotlin автоматически определяет тип каждой переменной.

Подобно тому как среда IntelliJ помогает поменять var на val, она может помочь убрать ненужное объявление типа. Щелкните на объявлении типа `String` (`: String`) рядом с `heroName` и нажмите Option-Return (Alt-Enter). Затем щелкните на строке Remove exclicit type specification в появившемся меню (рис. 2.6).

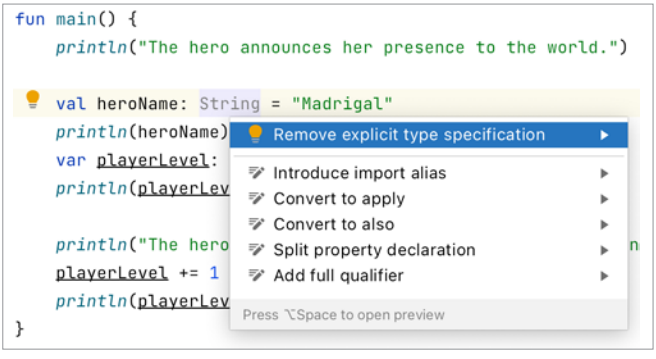

Рис. 2.6. Удаление явного определения типа

Объявление типа `: String` исчезнет. Повторите процесс для `playerLevel var`, чтобы убрать `: Int`.

Вне зависимости от того, используете вы автоматическое определение типов или указываете тип в объявлении каждой переменной, компилятор отслеживает тип. В этой книге мы используем автоматическое определение типов, если это не создает двусмысленности. Такой прием делает код более чистым и компактным и упрощает его изменение в будущем.

Обратите внимание, что IntelliJ покажет тип любой переменной по вашему запросу, даже если ее тип не был объявлен явно. Чтобы узнать тип переменной, щелкните на ее имени или выделите часть кода и выполните команду View ▸ Type Info (или нажмите Control-Shift-P [Ctrl-Shift-P]). Результат показан на рис. 2.7.

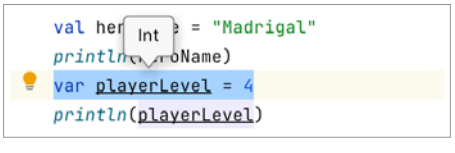

Рис. 2.7. Вывод информации о типе

Константы времени компиляции

Ранее мы рассказали о том, что `vars` могут менять свои значения, а `vals` нет. Мы немного приврали, но... из благих побуждений. На самом деле иногда `val` может возвращать разные значения, и мы обсудим это в главе 13. Если есть значения, которые вы не хотите менять никогда и ни при каких условиях, стоит использовать *константы времени компиляции*.

Константа времени компиляции объявляется вне какой-либо функции, даже не в пределах функции `main`, потому что ее значение присваивается *во время компиляции* (в момент, когда программа компилируется), — отсюда и такое название. Функция `main` и другие вызываются *во время выполнения* (когда программа за-

пущена), и переменные внутри функций получают свои значения в этот период. Константа времени компиляции к тому моменту уже существует.

Константы времени компиляции могут иметь значения только одного из следующих базовых типов (использование более сложных типов может поставить под угрозу гарантию времени компиляции). Вы узнаете больше о конструировании типов в главе 14. Итак, вот допустимые базовые типы для констант времени компиляции:

- `String`
- `Int`
- `Double`
- `Float`
- `Long`
- `Short`
- `Byte`
- `Char`
- `Boolean`

Имя героя игры (`Madrigal`) никогда не изменится — это главный персонаж, и у игрока нет возможности его сменить. Чтобы отразить эту неизменность в коде, можно задать значение имени константой. В файле Main.kt переместите переменную `heroName` над объявлением функции `main` и добавьте модификатор `const`.

Листинг 2.10. Объявление константы времени компиляции (Main.kt)

```
const val heroName = "Madrigal"

fun main() {
    println("The hero announces her presence to the world.")

    val heroName = "Madrigal"
    println(heroName)
    var playerLevel: Int = 4
    println(playerLevel)

    println("The hero embarks on her journey to locate the enchanted sword.")
    playerLevel += 1
    println(playerLevel)
}
```

Модификатор `const`, предшествующий `val`, предупреждает компилятор, что значение `val` нигде не должно изменяться. В данном случае имя героя всегда будет представлено значением `"Madrigal"`, что бы ни произошло. Это дает возможность компилятору применить дополнительные оптимизации. Снова выполните этот код и убедитесь, что результат остался прежним после внесения изменений.

В Kotlin принято использовать верблюжийРегистр (camelCase) для имен переменных и ЗМЕИНЫЙ_РЕГИСТР (SNAKE_CASE) для имен констант. Таким образом, вместо того чтобы присваивать новой константе имя `heroName`, правильнее было бы назвать ее `HERO_NAME`.

Так как наш проект невелик, вы можете легко изменить имя и обновить всего одно упоминание в функции `main`. Но в более крупных проектах эта задача заметно

усложняется. К счастью, IntelliJ включает средства рефакторинга, которые помогают вносить изменения на уровне всего кода.

Щелкните правой кнопкой мыши на константе `heroName`, затем выберите команду Refactor ▸ Rename... (рис. 2.8).

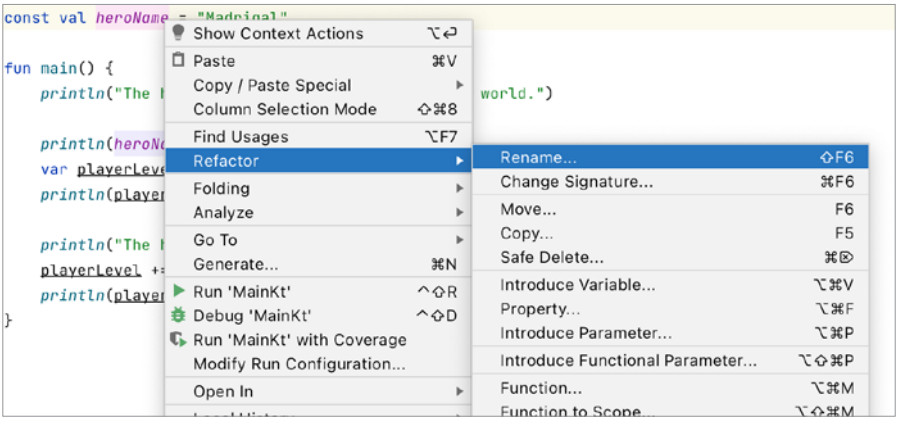

Рис. 2.8. Команда меню Refactor ▸ Rename...

Имя константы выделяется цветом. Введите `HERO_NAME`, заменяя выделенный текст. В процессе ввода IntelliJ продолжает выделять константу, как показано на рис. 2.9.

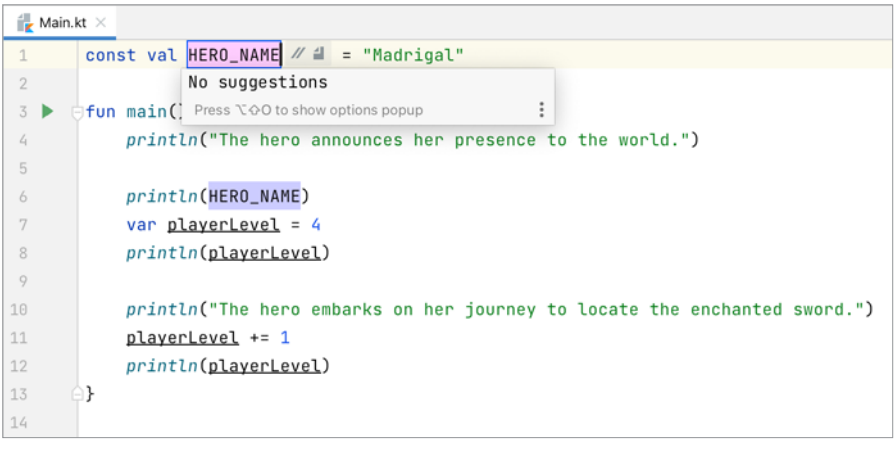

Рис. 2.9. Переименование константы

В процессе ввода обратите внимание, что при вводе нового имени строка `println(heroName)` автоматически обновляется. IntelliJ находит все случаи

употребления константы в вашем проекте и обновляет их, задавая новое имя. В проекте bounty-board замена затронет только одну строку кода, но в большом проекте IntelliJ может автоматически обновить множество файлов.

Когда вы завершите ввод нового имени, нажмите Return, чтобы подтвердить изменения.

Изучаем байт-код Kotlin

Из главы 1 вы узнали, что на Kotlin можно писать программы для виртуальной машины JVM, которая исполняет байт-код Java. Зачастую бывает полезно взглянуть на байт-код Java, который генерируется компилятором языка Kotlin и запускается под управлением JVM. Кое-где в этой книге мы будем рассматривать байт-код, чтобы понять, как конкретные особенности языка работают в JVM.

Умение анализировать Java-эквиваленты кода на Kotlin поможет вам понять, как работает Kotlin, особенно если у вас есть опыт работы с Java. Если опыта именно с Java у вас нет, вы все равно сможете увидеть в Kotlin знакомые черты языка, с которым вам приходилось работать, поэтому рассматривайте байт-код как своего рода псевдокод, упрощающий понимание. Ну а если вы новичок в программировании — поздравляем! Kotlin позволит вам выразить ту же логику программы, что и в Java, но гораздо короче.

Допустим, вам хочется узнать, как автоматическое определение типов переменных в Kotlin влияет на байт-код, сгенерированный для выполнения в JVM. Для этого воспользуйтесь инструментальным окном байт-кода Kotlin.

В файле Main.kt дважды нажмите клавишу Shift, чтобы открыть диалоговое окно Search Everywhere (поиск везде). Начните вводить: «show Kotlin bytecode» (показать байт-код Kotlin) и выберите из списка доступных действий Show Kotlin bytecode, как только оно появится (рис. 2.10).

Откроется инструментальное окно байт-кода Kotlin (рис. 2.11). (Также можно открыть его с помощью меню Tools ▸ Kotlin ▸ Show Kotlin Bytecode.)

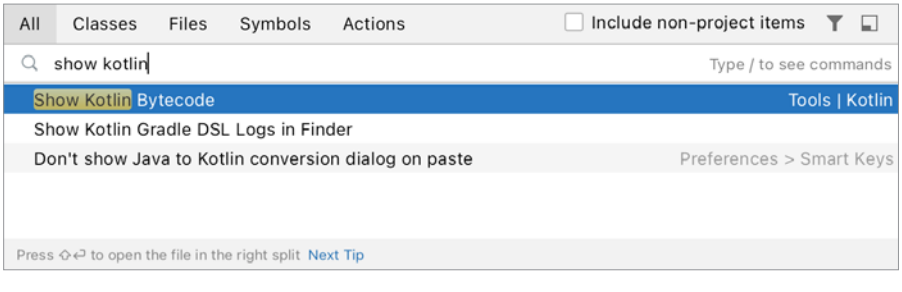

Рис. 2.10. Вывод байт-кода Kotlin

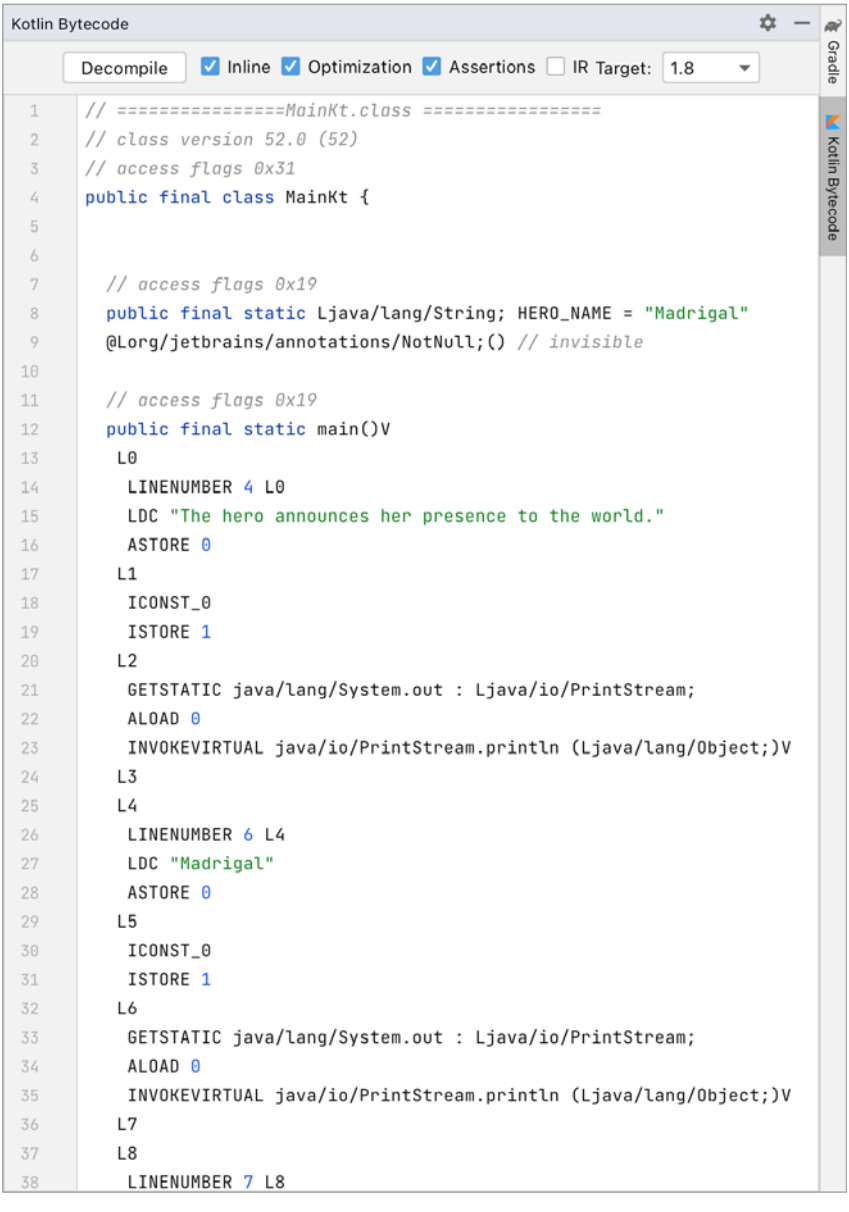

Рис. 2.11. Инструментальное окно байт-кода Kotlin

Если байт-код не ваш родной язык, не бойтесь! Переведите байт-код обратно в Java, чтобы увидеть его в более знакомом варианте. В окне байт-кода нажмите кнопку Decompile слева наверху.