1.5. Наследование. Интерфейсы и абстрактные классы

Сайт: Samsung Innovation Campus
Курс: Мобильная разработка на Kotlin
Книга: 1.5. Наследование. Интерфейсы и абстрактные классы

Напечатано:: Павел Степанов
Дата: вторник, 31 октября 2023, 13:05

Оглавление

1.5.1. Наследование. Интерфейсы и абстрактные классы

1.5.2. Подклассы

1.5.3. Иерархия типов в Kotlin

1.5.4. Умное приведение типов

1.5.5. Приведение типов

1.5.6. Объявление и реализация интерфейса

1.5.7. Реализация интерфейса по умолчанию

1.5.8. Абстрактные классы

1.5.1. Наследование. Интерфейсы и абстрактные классы

Интерфейсы в Kotlin очень похожи на интерфейсы в Java 8. Они могут содержать абстрактные методы, методы с реализацией. Главное отличие интерфейсов от абстрактных классов заключается в невозможности хранения переменных экземпляров. Они могут иметь свойства, но те должны быть либо абстрактными, либо предоставлять реализацию методов доступа.

Интерфейс определяется ключевым словом interface:

```
interface MyInterface {
    fun bar()
    fun foo() {
      // необязательное тело
    }
}
```

1.5.2. Подклассы

Подклассы

1.5.3. Иерархия типов в Kotlin

В Kotlin всё является объектом, в том смысле, что пользователь может вызвать функцию или получить доступ к свойству любой переменной. Некоторые типы являются встроенными, т.к. их реализация оптимизирована, хотя для пользователя они могут выглядеть как обычные классы. В данном разделе описывается большинство этих типов: числа, символы, логические переменные и массивы. Числа

Kotlin обрабатывает численные типы примерно так же, как и Java, хотя некоторые различия всё же присутствуют. Например, отсутствует неявное расширяющее преобразование для чисел, а литералы в некоторых случаях немного отличаются.

Для представления чисел в Kotlin используются следующие встроенные типы (подобные типам в Java): Тип Количество бит Double 64 Float 32 Long 64 Int 32 Short 16 Byte 8

Обратите внимание, что символы (characters) в языке Kotlin не являются числами (в отличие от Java). Символьные постоянные

В языке Kotlin присутствуют следующие виды символьных постоянных (констант) для целых значений:

Десятичные числа: 123 Тип Long обозначается заглавной L: 123L Шестнадцатеричные числа: 0x0F Двоичные числа: 0b00001011

ВНИМАНИЕ: Восьмеричные литералы не поддерживаются.

Также Kotlin поддерживает числа с плавающей запятой:

```
Тип Double по умолчанию: 123.5, 123.5e10
Тип Float обозначается с помощью f или F: 123.5f
```

Нижние подчеркивания в числовых литералах (начиная с версии 1.1)

Вы можете использовать нижние подчеркивания, чтобы сделать числовые константы более читаемыми:

```
val oneMillion = 1_000_000
val creditCardNumber = 1234_5678_9012_3456L
val socialSecurityNumber = 999_99_9999L
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
```

Open in Playground → Target: JVM Running on v.1.9.10

Представление

Обычно платформа Java хранит числа в виде примитивных типов JVM; если же нам необходима ссылка, которая может принимать значение null (например, Int?), то используются обёртки. В приведённом ниже примере показано использование обёрток.

Обратите внимание, что использование обёрток для одного и того же числа не гарантирует равенства ссылок на них:

```
val a: Int = 10000
print(a === a) // Prints 'true'
val boxedA: Int? = a
val anotherBoxedA: Int? = a
print(boxedA === anotherBoxedA) // !!!Prints 'false'!!!
```

Open in Playground → Target: JVM Running on v.1.9.10

Однако, равенство по значению сохраняется:

```
val a: Int = 10000
print(a == a) // Prints 'true'
val boxedA: Int? = a
val anotherBoxedA: Int? = a
print(boxedA == anotherBoxedA) // Prints 'true'
```

Open in Playground → Target: JVM Running on v.1.9.10

Явные преобразования

Из-за разницы в представлениях меньшие типы не являются подтипами бОльших типов. В противном случае у нас возникли бы сложности:

```
// Возможный код, который на самом деле не скомпилируется:
val a: Int? = 1 // "Обёрнутый" Int (java.lang.Integer)
val b: Long? = a // неявное преобразование возвращает "обёрнутый" Long (java.lang.Long)
print(a == b) // Данное выражение выведет "false" т. к. метод equals() типа Long предполагает, что вторая часть выражения также имеет тип Long
```

Open in Playground → Target: JVM Running on v.1.9.10

Таким образом, будет утрачена не только тождественность (равенство по ссылке), но и равенство по значению.

Как следствие, неявное преобразование меньших типов в большие НЕ происходит. Это значит, что мы не можем присвоить значение типа Bytепеременной типа Int без явного преобразования:

```
val b: Byte = 1 // порядок, литералы проверяются статически
val i: Int = b // ОШИБКА
```

Мы можем использовать явное преобразование для "расширения" чисел

Target: JVM Running on v.1.9.10

```
val i: Int = b.toInt() // порядок: число явно расширено
```

Open in Playground → Target: JVM Running on v.1.9.10

Каждый численный тип поддерживает следующие преобразования:

```
toByte(): Byte
    toShort(): Short
    toInt(): Int
    toLong(): Long
    toFloat(): Float
    toDouble(): Double
    toChar(): Char
```

Open in Playground → Target: JVM Running on v.1.9.10

Отсутствие неявного преобразования редко бросается в глаза, поскольку тип выводится из контекста, а арифметические действия перегружаются для подходящих преобразований, например:

```
val l = 1L + 3 // Long + Int => Long
```

Open in Playground → Target: JVM Running on v.1.9.10

Арифметические действия

Kotlin поддерживает обычный набор арифметических действий над числами, которые объявлены членами соответствующего класса (тем не менее, компилятор оптимизирует вызовы вплоть до соответствующих инструкций). См. Перегрузка операторов.

Что касается битовых операций, то вместо особых обозначений для них используются именованные функции, которые могут быть вызваны в инфиксной форме, к примеру:

```
val x = (1 shl 2) and 0x000FF000
```

Open in Playground → Target: JVM Running on v.1.9.10

Ниже приведён полный список битовых операций (доступны только для типов Int и Long):

```
shl(bits) – сдвиг влево с учётом знака (<< в Java)
shr(bits) – сдвиг вправо с учётом знака (>> в Java)
ushr(bits) – сдвиг вправо без учёта знака (>>> в Java)
and(bits) – побитовое И
or(bits) – побитовое ИЛИ
```

```
xor(bits) – побитовое исключающее ИЛИ
inv() – побитовое отрицание
```

Open in Playground → Target: JVM Running on v.1.9.10

Символы

Символы в Kotlin представлены типом Char. Напрямую они не могут рассматриваться в качестве чисел

```
fun check(c: Char) {
  if (c == 1) { // ОШИБКА: несовместимый тип
    // ...
  }
}
```

Open in Playground → Target: JVM Running on v.1.9.10

Символьные литералы записываются в одинарных кавычках: '1', '\n', '\uFF00'. Мы можем явно привести символ в число типа Int

```
fun decimalDigitValue(c: Char): Int {
  if (c !in '0'..'9')
    throw IllegalArgumentException("Вне диапазона")
  return c.toInt() - '0'.toInt() // Явные преобразования в число
}
```

Open in Playground → Target: JVM Running on v.1.9.10

Подобно числам, символы оборачиваются при необходимости использования nullable ссылки. При использовании обёрток тождественность (равенство по ссылке) не сохраняется. Логический тип Тип Boolean представляет логический тип данных и принимает два значения: true и false. При необходимости использования nullable ссылок логические переменные оборачиваются. Встроенные действия над логическими переменными включают || – ленивое логическое ИЛИ && – ленивое логическое И ! - отрицание

Массивы Массивы в Kotlin представлены классом Array, обладающим функциями get и set (которые обозначаются [] согласно соглашению о перегрузке операторов), и свойством size, а также несколькими полезными встроенными функциями:

```
class Array private constructor() {
  val size: Int
  fun get(index: Int): T
  fun set(index: Int, value: T): Unit
```

Open in Playground → Target: JVM Running on v.1.9.10

fun iterator(): Iterator // ... } Для создания массива мы можем использовать библиотечную функцию arrayOf(), которой в качестве аргумента передаются элементы массива, т.е. выполнение arrayOf(1, 2, 3) создаёт массив [1, 2, 3]. С другой стороны библиотечная функция arrayOfNulls() может быть использована для создания массива заданного размера, заполненного значениями null.

Также для создания массива можно использовать фабричную функцию, которая принимает размер массива и функцию, возвращающую начальное значение каждого элемента по его индексу:

```
// создаёт массив типа Array со значениями ["0", "1", "4", "9", "16"]
val asc = Array(5, { i -> (i * i).toString() })
```

Open in Playground → Target: JVM Running on v.1.9.10

Как отмечено выше, оператор [] используется вместо вызовов встроенных функций get() и set().

Обратите внимание: в отличие от Java массивы в Kotlin являются инвариантными. Это значит, что Kotlin запрещает нам присваивать массив Array переменной типа Array, предотвращая таким образом возможный отказ во время исполнения (хотя вы можете использовать Array, см. Проекции типов).

Также в Kotlin есть особые классы для представления массивов примитивных типов без дополнительных затрат на оборачивание: ByteArray, ShortArray, IntArray и т.д. Данные классы не наследуют класс Array, хотя и обладают тем же набором методов и свойств. У каждого из них есть соответствующая фабричная функция:

```
val x: IntArray = intArrayOf(1, 2, 3)
x[0] = x[1] + x[2]
```

Open in Playground → Target: JVM Running on v.1.9.10

Строки

Строки в Kotlin представлены типом String. Строки являются неизменяемыми. Строки состоят из символов, которые могут быть получены по порядковому номеру: s[i]. Проход по строке выполняется циклом for:

```
for (c in str) {
  println(c)
}
```

Open in Playground → Target: JVM Running on v.1.9.10

Строковые литералы

В Kotlin представлены два типа строковых литералов: строки с экранированными символами и обычные строки, которые могут содержать символы новой строки и произвольный текст. Экранированная строка очень похожа на строку в Java:

```
val s = "Hello, world!\n"
```

Open in Playground → Target: JVM Running on v.1.9.10

Экранирование выполняется общепринятым способом, а именно с помощью обратной косой черты.

Обычная строка выделена тройной кавычкой ("""), не содержит экранированных символов, но может содержать символы новой строки и любые другие символы:

```
val text = """
  for (c in "foo")
    print(c)
"""
```

Open in Playground → Target: JVM Running on v.1.9.10

Строковые шаблоны

Строки могут содержать шаблонные выражения, т.е. участки кода, которые выполняются, а полученный результат встраивается в строку. Шаблон начинается со знака доллара ($) и состоит либо из простого имени (например, переменной):

```
val i = 10
val s = "i = $i" // evaluates to "i = 10" либо из произвольного выражения в фигурных скобках:
val s = "abc"
val str = "$s.length is ${s.length}" // evaluates to "abc.length is 3"
```

Open in Playground → Target: JVM Running on v.1.9.10

Шаблоны поддерживаются как в обычных, так и в экранированных строках. При необходимости символ $ может быть представлен с помощью следующего синтаксиса:

```
val price = "${'$'}9.99"
```

Open in Playground → Target: JVM Running on v.1.9.10

1.5.4. Умное приведение типов

Во многих случаях в Kotlin вам не нужно использовать явные приведения, потому что компилятор следит за is-проверками для неизменяемых значений и вставляет приведения автоматически, там, где они нужны:

```
fun demo(x: Any) {
    if (x is String) {
        print(x.length) // x автоматически преобразовывается в String
    }
}
```

Open in Playground → Target: JVM Running on v.1.9.10

Компилятор достаточно умён для того, чтобы делать автоматические приведения в случаях, когда проверка на несоответствие типу (!is) приводит к выходу из функции:

```
if (x !is String) return
print(x.length) // x автоматически преобразовывается в String
```

Open in Playground → Target: JVM Running on v.1.9.10

или в случаях, когда приводимая переменная находится справа от оператора && или ||:

```
// x автоматически преобразовывается в String справа от `||`
if (x !is String || x.length == 0) return

// x автоматически преобразовывается в String справа от `&&`
if (x is String && x.length > 0) {
    print(x.length) // x автоматически преобразовывается в String
}
```

Open in Playground → Target: JVM Running on v.1.9.10

Такие умные приведения работают вместе с when-выражениями и циклами while:

```
when (x) {
    is Int -> print(x + 1)
    is String -> print(x.length + 1)
    is IntArray -> print(x.sum())
```

```
    }
```

Заметьте, что умные приведения не работают, когда компилятор не может гарантировать, что переменная не изменится между проверкой и использованием. Более конкретно, умные приведения будут работать:

- с локальными val переменными - всегда;
- с val свойствами - если поле имеет модификатор доступа private или internal, или проверка происходит в том же модуле, в котором объявлено это свойство. Умные приведения неприменимы к публичным свойствам или свойствам, которые имеют переопределённые getter'ы;
- с локальными var переменными - если переменная не изменяется между проверкой и использованием и не захватывается лямбдой, которая её модифицирует;
- с var свойствами - никогда (потому что переменная может быть изменена в любое время другим кодом).

1.5.5. Приведение типов

Операторы is и !is

Мы можем проверить принадлежит ли объект к какому-либо типу во время исполнения с помощью оператора is или его отрицания !is:

```
if (obj is String) {
    print(obj.length)
}

if (obj !is String) { // то же самое, что и !(obj is String)
    print("Not a String")
}
else {
    print(obj.length)
}
```

Open in Playground → Target: JVM Running on v.1.9.10

Нередко может возникать задача по преобразованию типов, например, чтобы использовать данные одного типа в контексте, где требуются данные другого типа. В этом случае Kotlin представляет ряд возможностей по преобразованию типов. Встроенные методы преобразования типов

Для преобразования данных одного типа в другой можно использовать встроенные следующие функции, которые есть у базовых типов (Int, Long, Double и т.д.):

```
toByte
toShort
toInt
toLong
toFloat
toDouble
toChar
```

Open in Playground → Target: JVM Running on v.1.9.10

Все эти функции преобразуют данные в тот тип, которые идет после префикса to:

```
toByte.
val s: String = "12"
val d: Int = s.toInt()
println(d)
```

Open in Playground → Target: JVM Running on v.1.9.10

В данном случае строка s преобразуется в число d. Просто так передать строку переменной типа Int, мы не можем, несмотря на то, что вроде бы строка и содержит число 12:

```
val d: Int = "12"   // ! Ошибка
```

Open in Playground → Target: JVM Running on v.1.9.10

Однако надо учитывать, что значение не всегда может быть преобразовано к определенному типу. И в этом случае генерируется исключение. Соответственно в таких случаях желательно отлавливать исключение:

```
val s: String = "tom"
try {
    val d: Int = s.toInt()
    println(d)
}
catch(e: NumberFormatException){
    println(e.message)
}
```

Open in Playground → Target: JVM Running on v.1.9.10

1.5.6. Объявление и реализация интерфейса

Интерфейсы в Kotlin очень похожи на интерфейсы в Java 8. Они могут содержать абстрактные методы, методы с реализацией. Главное отличие интерфейсов от абстрактных классов заключается в невозможности хранения переменных экземпляров. Они могут иметь свойства, но те должны быть либо абстрактными, либо предоставлять реализацию методов доступа.

Интерфейс определяется ключевым словом interface:

```
interface MyInterface {
    fun bar()
    fun foo() {
      // необязательное тело
    }
}
```

Open in Playground → Target: JVM Running on v.1.9.10

Реализация интерфейсов

Класс или объект могут реализовать любое количество интерфейсов:

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

Open in Playground → Target: JVM Running on v.1.9.10

Свойства в интерфейсах

Вы можете объявлять свойства в интерфейсах. Свойство, объявленное в интерфейсе, может быть либо абстрактным, либо иметь свою реализацию методов доступа. Свойства в интерфейсах не могут иметь теневых полей, соответственно, методы доступа к таким свойствам не могут обращаться к теневым полям.

```
interface MyInterface {
    val prop: Int // абстрактное свойство
    val propertyWithImplementation: String
        get() = "foo"
    fun foo() {
```

```
        print(prop)
    }
}
class Child : MyInterface {
    override val prop: Int = 29
}
```

Open in Playground →

Target: JVM Running on v.1.9.10

1.5.7. Реализация интерфейса по умолчанию

Устранение противоречий при переопределении

Когда мы объявляем большое количество типов в списке нашего супертипа, может так выйти, что мы допустим более одной реализации одного и того же метода. Например:

```
interface A {
    fun foo() { print("A") }
    fun bar()
}
interface B {
    fun foo() { print("X") }
    fun bar() { print("bar") }
}
class C : A {
    override fun bar() { print("bar") }
}
    override fun foo() {
        super<A>.foo()
        super<B>.foo()
    }
    override fun bar() {
        super<B>.bar()
    }
```

Оба интерфейса `A` и `B` объявляют функции `foo()` и `bar()`. Оба реализуют `foo()`, но только `B` содержит реализацию `bar()` (`bar()` не отмечен как абстрактный метод в интерфейсе `A`, потому что в интерфейсах это подразумевается по умолчанию, если у функции нет тела). Теперь, если мы унаследуемся каким-нибудь классом `C` от интерфейса `A`, нам, очевидно, придётся переопределять метод `bar()`, обеспечивая его

реализацию. Однако, если мы унаследуемся классом `D` от интерфейсов `A` и `B`, нам надо будет переопределять все методы, которые мы унаследовали от этих интерфейсов. Это правило касается как тех методов, у которых имеется только одна реализация (`bar()`), так и тех, у которых есть несколько реализаций (`foo()`).

1.5.8. Абстрактные классы

Класс и некоторые его члены могут быть объявлены как `abstract`. Абстрактный член не имеет реализации в своём классе. Обратите внимание, что нам не надо аннотировать абстрактный класс или функцию словом open - это подразумевается и так.

Можно переопределить неабстрактный open член абстрактным

```
open class Polygon {
    open fun draw() {}
}

abstract class Rectangle : Polygon() {
    abstract override fun draw()
}
```

Open in Playground → Target: JVM Running on v.1.9.10

Начать тур для пользователя на этой странице