Kotlin

Василий Купоросов

29 июня 2019 г.

Содержание

1. Базовый синтаксис

1.1. Операторы

1.1.1. `if`

В отличие от джавы, в котлине оператор `if` является выражением, то есть имеет тип и возвращает результат. Его можно передавать как аргумент в функцию, присваивать в переменную, итд. Тернарного оператора в котлине нет, но вместо него как раз хорошо подойдет `if`.

```
val maximum = if (3 > 5) ''three'' else ''five''
```

Если выражения в разных ветках оператора имеют разные типы, то у результата будет наиболее общий тип.

1.1.2. `for`

Обычного плюсового оператора `for(;;)` в котлине нет, но есть аналог C#-ого `foreach`.

```
for (i in 1..10) {
  println(i)
}
```

Итерироваться можно по объектам, которые имеют оператор `iterator`. Примеры стандартных итерируемых конструкций – диапазоны.

```
for (i in 1..n) // for (i = 1; i <= n; ++i)
for (i in 4..1) // выполнится ноль итераций, т.к. 4 > 1
for (i in 4 downTo 1) // for (i = 4; i >= 1; -i)
for (i in 1..4 step 2) // for (int i = 1; i <= 4; i += 2)
for (i in 4 downTo 1 step 2) // for (int i = 4; i >= 1; i -= 2)
for (i in 1 until 4) // for (int i = 1; i < 4; ++i)
```

Такие диапазоны можно использовать и с оператором `if`.

```
if (i in 1..n) // if (1 <= i && i <= n)
```

1.1.3. `when`

`when` – это аналог `switch`, только позволяющий делать более сложные проверки и тоже являющийся выражением. В отличие от `switch`, в `when` выполняться будет ровно один кейс, а не все начиная с подошедшего.

```
//kotlin:
when (x) {
  1 -> println(''x == 1'')
  2 -> println(''x == 2'')
  else -> {
    println(''x is neither 1 nor 2'')
  }
}
```

```
//java:
switch (x) {
  case 1:
    System.out.println("x == 1");
    break;
  case 2:
    System.out.println("x == 2");
    break;
  default:
    System.out.println("x is neither 1 nor 2");
}
```

Если возвращаемое значение оператора `when` будет использоваться (для присваивания в переменную или передачи в функцию), то внутри обязательно писать `else`.

```
val s = when (x) { // Ошибка компиляции: выражение 'when' должно быть исчерпывающим
  1 -> "one"
  2 -> "two"
}
```

```
when (x) { // OK
  1 -> println("one")
  2 -> println("two")
}
```

```
val s = when (x) { // OK
  1 -> "one"
  2 -> "two"
  else -> "many"
}
```

Условия можно объединять.

```
when (x) {
  1, 2 -> println("x == 1 or x == 2")
  else -> println("x != 1 and x != 2")
}
```

Как и в `if`, можно использовать оператор `in` для проверки принадлежности диапазону.

```
when (x) {
  in 1..10 -> println("x in [1; 10]")
  !in 10..20 -> println("x in outside [10; 20]")
  else -> println("other")
}
```

Можно использовать `when` без проверяемого объекта.

```
when {
  x.isOdd() -> println("x is odd")
  x.isEven() -> println("x is even")
  else -> println("x is strange")
}
```

Так как `when` является выражением, его результат можно передать в качестве аргумента или вернуть из функции.

```
fun countLetters(str: String): String {
  return when (str.length) {
    1 -> ''one''
    2 -> ''two''
    else -> ''many''
  }
}
```

1.2. Переменные

Для объявления неизменяемых переменных используется ключевое слово `val`, для изменяемых – `var`. Типы могут выводиться автоматически. Если же требуется явно указать тип, это можно сделать через двоеточие.

```
val a = 42 // выведется тип Int
val b: Int = a
a += 1 // ошибка компиляции, a - read-only
var s = ''foo''
s = ''bar'' // OK
```

1.3. Функции

Функции объявляются ключевым словом `fun`. Если функция состоит из одного выражения, ее можно записать через = и не указывать тип. Также тип можно не указывать, если функция ничего не возвращает.

```
fun sum(a: Int, b: Int) = a + b
fun inc(a: Int): Int = a + 1
fun printHello() {
  println(''Hello!'')
}
```

1.4. Свойства

В джаве считается, что публичные поля использовать плохо, поэтому в котлине полей нет вообще. Зато есть свойства. Свойство можно объявить на уровне файла или внутри класса. При компиляции в байт-код, для него автоматически сгенерируются геттер и сеттер. То есть с точки зрения джавы, это будет приватное поле. Также геттеры и сеттеры можно переопределить.

```
class Kokoko {
  val className: String
    get() = ''Ko-ko-ko class''

}
```

Такой подход лучше публичных полей, потому что, если вдруг понадобится добавить в геттер или сеттер более сложную логику (логирование, кеширование, итд), то не придется менять менять код в тех местах, где было обращение к свойству. То есть с точки зрения использование это всё еще будет выглядеть как просто обращение к полю класса.

```
class Kokoko {
  val className: String
    get() = {
      println(''property className has been used'')
      return ''Ko-ko-ko class''
    }
}
...
val kokoko = Kokoko() // создали объект класса Kokoko
val name = kokoko.className // в stdout напечатается ''property className has been used''
```

Возникает вопрос, когда лучше всего использовать `val` и кастомный `getter`, а когда просто метод.

```
val className: String
  get() = ''Ko-ko-ko class''

fun getName(): String = ''Ko-ko-ko class''
```

Для этого придумали конвенцию, что для выстрых вычислений ($\mathcal{O}(1)$) без сайд-эффектов используется `val + getter`, а для более содержательных вычислений – функция.

1.5. Классы

Классы в котлине могут содержать свойства, функции и конструкторы, есть наследование.

```
interface SuperInterface {
  val value: Int
  fun getName() = ''SI''
}

class MyClass : SuperInterface {
  override val value: Int
  var myName: String
  override fun getName() = ''MC''
}
```

1.5.1. Первичный конструктор

Первичный конструктор позволяет не писать каждый раз конструкции такого вида:

```
// Java
class MyClass {
  int x;
  int y;

  MyClass(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

Вместо этого первичный конструктор создает проперти, которые инициализируются параметрами, передаваемыми при создании объекта и живут в нем до его уничтожения.

```
class MyClass(val x: Int, val y: Int) { ... }
...
val myVar = MyClass(3, 4)
println(myVar.x) // выведется '3'
```

1.5.2. Вторичные конструкторы

Вторичные – это конструкторы, объявленные в теле класса. Они могут выполнять какую-то более сложную логику при создании объекта (например скачать ресурс по сети).

```
class Resource {
  val data: String

  constructor(url: String) {
    data = downloadFromUrl(url)
  }
}
```

1.6. Модификаторы

1.6.1. Visibility

Модификаторы видимости бывают `public`, `private`, `protected` и `internal` (`package-private`). По умолчанию видимость всегда `public`.

1.6.2. Modality

Modality-модификаторы бывают `abstract`, `open` и `final`. По умолчанию все классы, свойства и функции имеют модификатор `final` (кроме интерфейсов и их содержимого). Это значит, что по умолчанию нельзя наследоваться от не абстрактных классов и переопределять методы. Чтобы это можно быть сделать, надо использовать модификатор `open`. Модификатор `final` для пропертей означает, что их нельзя переопределить в классах наследниках (чтобы пропертя была иммутабельной, ее надо объявить ключевым словом `val`).

```
class FinalBaseClass { ... }

open class OpenBaseClass {
  var x: Int = 42
  fun finalFoo() { ... }
  open fun openFoo() { ... }
}

class MyClass : FinalBaseClass() { ... } // Ошибка компиляции, наследование от финального класса

class MyClass : OpenBaseClass() {
  override var x: Int = 43 // Ошибка компиляции, переопределение финальной проперти
  override fun finalFoo() { ... } // Ошибка компиляции, переопределение финальной функции
  override fun openFoo() { ... } // OK
}
```

1.7. `data class`

Часто мы хотим завести класс, который будет просто хранить какие-то значения. При этом для него надо писать конструктор, `equals`, `hashCode`, `toString`, и при добавлении новых свойств, не забывать это всё менять. Для удобства были введены `data`-классы, которые делают это автоматически.

```
// Котлин:
data class Client(val name: String, val email: String)
```

```
// Java:
public class Client {
  private final String name;
  private final String email;
  public Client(String name, String email) {
    this.name = name;
    this.email = email;
  }
  public String getName() { return name; }
  public String getEmail() { return email; }
  public boolean equals(Client other) { ... }
  public int hashCode() { ... }
  public String toString() { ... }
}
```

1.8. Объекты

1.8.1. `object`

В котлине можно создать синглтон-объект с помощью ключевого слова `object`. У такого объекта не может быть конструктора, так как он создается при первом обращении к нему.

```
object MySingleton : SuperInterface {
  override val value = 0
  val name = ''MySingleton''
}
```

1.8.2. `companion object`

В котлине нет статических свойств и функций, но есть `companion`-объекты. Всё что объявлено внутри такого объекта, становится доступным без экземпляра самого класса, то есть статическим.

```
class MyClass {
  companion object : SuperInterface {
    var staticVar = 42
    fun staticFoo(x: Int) {
      staticVar = x
    }
  }
}

println(MyClass.staticVar) // 42
MyClass.staticFoo(24) // MyClass.staticVar = 24
```

Также `companion object` можно передать в функцию.

```
fun foo(obj: SuperInterface) { ... }
...
foo(MyClass.Companion) // MyClass.Companion имплементирует интерфейс SuperInterface
                       // (см пример выше)
```

Тут снова возникает дилемма. Вместо того чтобы использовать `companion`-объекты, можно вынести свойства и функции из метода на уровень файла. По конвенции объекты используют только если их надо куда-то передать, или если того требует инкапсуляция (textttcompanion-объекты могут обращаться к приватным методам класса). В остальных случаях лучше обходиться без них.

1.9. Интерполяция строк

Внутри строки можно написать `$`, а после него переменную или выражение в фигурных скобках. Тогда на это место подставится значение этой переменной или выражения.

```
fun foo(name: String, age: Int) {
  println("$name is ${ if (age < 30) "young" else "old" }")
}
```

1.10. Hello, world!

Чтобы написать программу, которую можно запустить, достаточно в каком-нибудь файле вне класса определить функцию со специальной сигнатурой.

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

Также можно написать `main` в объекте или в `companion object`.

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

Аннотация `@JvmStatic` говорит компилятору, что при трансляции в байт-код надо сделать честный статический метод. Такая аннотация может быть только у членов объектов и `companion`-объектов.

2. Типы

2.1. Базовые типы

В котлине разделения на примитивные ссылочные и типы с точки зрения синтаксиса нет. Базовые типы: `Int`, `Byte`, `Short`, `Long`, `Char`, `Double`, `Float` при компилации в байт-код, везде где нужно превратятся в соответствующие `boxed` или `unboxed` типы.

2.2. Массивы

Для базовых типов реализованы базовые массивы: `IntArray, ByteArray, ...`. Для массивов остальных типов существует `generic`-класс `Array<T>`. Заметим, что `IntArray != Array<Int>`, так как первый с точки зрения джавы хранит `int`, а второй – `Integer`.

2.3. Специальные типы

2.3.1. `Nothing`

Тип `Nothing` (bottom-тип) не может принимать никакого значения и используется в конструкциях типа `throw` или `return`, то есть в местах, где прерывается выполнения программы.

2.3.2. `Any`

`Any` (top-тип)– аналог `Object` в джаве.

2.3.3. `Unit`

В каком-то смысле `Unit` – это аналог `Void` в джаве. По факту это синглтон, который при компиляции транслируется в `Void`. `Unit` показывает, например, что функция ничего не возвращает. Еще его можно передать как `generic`-аргумент.

```
fun printHello(): Unit {
  println(''Hello!'')
}
```

2.3.4. Функциональные типы и лямбды

Функциональные типы – это аналоги `Function`, `Function2`, итд.

```
val foo: (Int, String) -> Double
// аналогично
val foo: Function2<Int, String, Double>
```

Лямбды в котлине такие же как в джаве, только синтаксис немного другой.

```
val positives = list.filter { x-> x > 0 }
val positives = list.filter { it > 0 } // параметр указывать не обязательно, для этого создается
                                       // переменная it
```

2.4. Nullable типы

Для любого типа T определен тип T?, который может принимать либо значение типа T, либо null. При этом T будет наследником T?. У таких типов запрещено вызывать функции, чтобы обезопаситься от NullPointerException'ов. Сам же null имеет тип Nothing?.

Есть несколько способов вызвать метод у Nullable-типа. Первый – проверить, что объект – не null. В этом случае произойдет "smart cast" переменной типа T? к типу T.

```
val x: Int? = parseInt(strX)
val y: Int? = parseInt(strY)
// нельзя написать x * y, потому что они типа Int?
if (x != null && y != null) {
  print(x * y) // OK, проверили, что не null
}
```

Также есть специальные операторы ?., ?: и !!.

```
val x: Int? = null

x?.inc() // if (x != null) x.inc() else null
x?.inc() ?: 0 // if (x != null) x.inc() else 0
x!!.inc() // if (x != null) x.int() else throw NullPointerException()
```

2.5. Use-site variance

В джаве есть wildcard'ы. В котлине они тоже есть и задаются ключевыми словами in и out.

```
// void copy(Array<? extends CharSequence> x, Array<? super CharSequence> y)
fun copy(x: Array<out CharSequence>, y: Array<in CharSequence>) {
  for (i in x.indices) {
    y[i] = x[i]
  }
}
// копируем из массива String в массив Any
fun foo(a: Array<String>, b: Array<Any>) {
  copy(a, b)
}
```

2.6. Declaration-site variance

Вайлдкарды можно также использовать и при объявлении класса. Это будет означать, что данный тип можно будет использовать либо только для аргументов функций, либо только для возвращаемого значения, а также позволит кастовать generic-объекты друг к другу.

```
interface List<out T> {
  fun get(index: Int): T // OK, T находится в out позиции
  fun set(index: Int, t: T) // Ошибка компиляции, T нельзя использовать в in позиции
}

fun bar(x: List<CharSequence>) { ... }
fun foo(x: List<String>) {
  bar(x) // Прикастовали List<String> к List<CharSequence>
}
```

2.7. Еще немного про вариантность

Пусть у нас есть интерфейсы `A, B, C`, наследующиеся по цепочке, шаблонный класс `Inv<T>` с инвариантным параметром и функция, принимающая `Int<B>`.

```
interface A
interface B : A
interface C : B

class Inv<T>

fun foo(i: Inv<B>) { }
```

Что мы можем передать в эту функцию?

```
fun test() {
  foo(Inv<A>()) // error
  foo(Inv<B>()) // ok
  foo(Inv<C>()) // error
}
```

В этом случае можем передать только `Inv<B>`, потому что шаблонный параметр в `Inv<B>` должен совпадать при объявлении и вызове. Тип, соответствующий такому классу, называется **инвариантным**.

Если же мы хотим передавать всех предков или всех потомков, надо добавить вариантность.

```
fun fooIn(i: Inv<in B>) { }
fun fooOut(i: Inv<out B>) { }

fun test() {
  fooIn(Inv<A>()) // ok
  fooIn(Inv<B>()) // ok
  fooIn(Inv<C>()) // error

  fooOut(Inv<A>()) // error
  fooOut(Inv<B>()) // ok
  fooOut(Inv<C>()) // ok
}
```

Класс, куда в качестве шаблонного параметра можно передавать любого предка (`<in T>`), называется **контрвариантным**, а если любого потомка (`<out T>`), то **ковариантным**.

Если вариантность задается при объявлении переменной (`fun fooIn(i: Inv<in B>)  `), это называется **use site variance**. Если при объявлении класса (`class In<in B>`), то **declaration site variance**.

Рассмотрим пример с ковариантным классом.

```
@Suppress("TYPE_VARIANCE_CONFLICT")
class Out<out T>(var value: T)

fun test() {
  val oString: Out<String> = Out("asd")
  val oAny: Out<Any> = oString // ok, String extends Any
  oAny.value = 42 // ok, oAny.value is Any
  val str: String = oString.value // ClassCastException, oString.value is Int
}
```

Здесь мы присваиваем класс, содержащий строчку, в класс, содержащий что угодно, потому что можем. Затем мы меняем строчку на число и получаем проблему, так как не можем скастить

содержимое `oString` к `String`.

Если убрать `Suppress`, Компилятор будет ругаться на этот код. А именно, когда мы заводим в классе `Out` поле `value`, для него генерируются геттер и сеттер. При этом ковариантный тип `T` в сеттере находится в позиции параметра. Это как раз приводит к описанной в примере проблеме, поэтому по-умолчанию такой класс не скомпилируется.

2.8. Smart casts

Пусть у нас есть переменная типа `Any`, мы хотим проверить, какого она на самом деле типа и вызвать соответствующие методы. В джаве мы бы сделали это так.

```
if (obj instanceof String) {
  String str = (String) obj;
  ...
}
```

В котлине не нужно создавать новый объект, потому что мы уже проверили, что тип нам подходит.

```
fun getStrLen(s: Any): Int? {
  if (str is String) {
    // str скастился к String
    return str.length
  }
  //str снова Any
  return null
}
```

То же самое можно делать внутри `when`.

```
fun hasPrefix(x: Any) = when(x) {
  is String -> x.startsWith('prefix')
  else -> false
}
```

2.9. Sealed classes

`sealed`-классы – это алгебраические классы, являющиеся, в каком-то смысле, расширением `enum`'ов (в котлине `enum`'ы такие же как в джаве).

Ключевое слово `sealed` при объявлении класса говорит компилятору, что все объекты, имеющие тип данного класса, могут быть только наследниками этого класса. При этом сам класс становится абстрактным и не финальным. Наследники `sealed`-классов могут быть объявлены только в том же файле, что и сам класс. Это позволяет убедиться, что других наследников точно не будет.

За счет накладываемых ограничений, мы можем упрощать некоторые проверки, например, не писать ветку `else` в операторе `when`, если ветки со всеми наследниками уже есть.

```
sealed class Expr
data class Const(val number: Double) : Expr()
data class Sum(val x: Expr, val y: Expr) : Expr()
object NotANumber : Expr()

fun eval(expr: Expr): Double = when(expr) {
  is Const -> expr.number
  is Sum -> eval(expr.x) + eval(expr.y)
  NotANumber -> Double.NaN // здесь не пишется is, потому что NotANumber - это объект
  // else не нужен, так как перебрали все варианты
}
```

2.10. Read-only и Mutable коллекции

В котлине есть все те же коллекции что и в джаве, но они бывают изменяемые и неизменяемые.

```
val lst: List<Int> = listOf(1, 2, 3) // Неизменяемый список
lst.add(4) // Ошибка

val m = lst.toMutableList()
m.add(4) // OK
```

В компиляторе для этого сделано примерно так – для всех стандартных коллекций есть свои интерфейсы без модифицирующих методов, а от них унаследованы интерфейсы, добавляющие эти методы.

```
interface MutableList<T> : List<T> {
  fun set(index: Int, t: T)
  ...
}
```

3. Разные виды функций

3.1. Анонимные функции

Анонимная функция – это функциональная переменная. Объявляем функцию, принимающую функциональный аргумент и передаем в неё анонимную функцию.

```
fun foo(block: (Int) -> String) = block(123) // вызываем переданную функцию от 123

fun test() {
  foo(fun(x: Int): String = x.toString()) // нормальное объявление функции только без имени
  foo({ x: Int -> x.toString() }) // в виде лямбды
  foo({ x -> x.toString() }) // можем не указывать тип
  foo({ it.toString() }) // если аргумент один, он будет доступен по идентификатору it
  foo { it.toString() } // если лямбда - последний аргумент, ее можно вынести за круглые скобки
  foo(Int::toString) // ссылка на функцию, как в Java
}
```

Преимущество функционального литерала (первый пример) над лямбдой в том, что можно явно указать тип возвращаемого значения.

3.2. Расширения

3.2.1. Extension functions

Функции-расширения позволяют расширять функционал класса без наследования и писать красивый код для всяких утилит. Сравним вызов `swap()` двух элементов в джаве и в котлине.

```
// java
Collections.swap(list, Collections.binarySearch(list, Collections.max(otherList)),
    Collections.max(list));
// kotlin + extension functions
list.swap(list.binarySearch(otherList.max()), list.max)
```

Расширения можно делать (и уже сделаны) даже для стандартных типов.

```
42.toBigInteger() // BigInteger(42)
''abc''.drop(1) // удаляем первый символ
''123.4''.toFloat() // парсим Float из строки
```

Чтобы объявить функцию-расширение, надо написать название класса и через точку название функции. Переменная, от которой вызовется функция, будет доступна в теле через идентификатор `this`. При этом тип, для которого объявляется расширение, называется ресивер (Receiver type).

```
fun String.popFront1() {
  this.drop(1)
}

fun String.popFront2() {
  drop(1) // this можно не писать
}
```

Extension функции являются синтаксическим сахаром для статических функций, принимающих экземпляр класса. То есть независимо от динамического типа, вызовется функция у класса, который был известен при компиляции.

```
open class C
class D : C()
fun C.foo() = ”c”
fun D.foo() = ”d”

fun printFoo(c: C) {
  println(c.foo())
}

printFoo(D()) // передаем экземпляр D, но выведется ”c”.
```

3.2.2. Member extensions

Extension функции можно объявлять внутри класса, чтобы уменьшить область видимости или получить доступ к приватным членам этого класса. В таком случае в теле этой функции будет несколько `this`'ов. Обращаться к ним надо через метки.

```
class A {
  fun String.foo() {
    this.length // равносильно this@foo.length
    this@A.bar()
  }

  fun bar() { }
}
```

3.2.3. Extension properties

Расширения могут также быть пропертями. Но поле для них генерироваться не будет. Поэтому надо явно написать геттер. Можно написать и сеттер, но, так как никакое поле не создастся, смотреться такой сеттер будет странно.

```
val String.numberOfA
  get() = count { it == 'A' } // считаем количество 'A'
  set(val) = println(”can not assign to an extension property”)

val String.abc = 1 // error
```

3.2.4. Generic extensions

Расширения также можно задавать на дженериках. Хороший пример – функция из стандартной библиотеки `let()`. Он хорошо подходит, например, для обработки случаев, когда результат какого-то выражение может быть null.

```
fun <T, R> T.let(block: (T) -> R): R = block(this)

a.nullableMethod()?.let { it.length } ?: 0 // если результат метода не null,
                                           // возвращаем его длину, иначе 0
```

3.2.5. Member vs Extension

Что вызовется, если внутри класса объявлена функция (member function), и есть функция-расширение с такой же сигнатурой? Ответ – всегда вызовется мембер-функция. Это позволяет, например избежать проблем, когда код работал, потом сделали импорт, содержащий расширение, и всё сломалось. Однако есть исключение – extension-функция `List.forEach()`. В котлине эту функцию переписали, и вызывается всегда она.

3.3. Операторные конвенции

3.3.1. Перегрузка операторов

В C++ есть перегрузка операторов. Ее проблема в том, что можно переопределить практически любую последовательность символов, а потом начнется путаница. Поэтому в джаве возможности перегружать операторы нет. В котлине пошли на компромисс и разрешили перегружать только некоторые основные операторы.

Чтобы перегрузить оператор для класса, надо написать соответствующую `extension`-функцию с ключевым словом `operator` для этого класса. Какие операторы можно перегружать.

- `unaryMinus`, `unaryPlus`, `not`
- `plus`, `minus`, `div`, `rem`, `times`
- `inc`, `dec`
- `plusAssign`, `minusAssign`, ... (`+=`, `-=`, ...)
- `set`, `get` (доступ к элементам через `[]`)
- `compareTo` (сразу все операторы сравнения)
- `contains` (`a in myList`)

3.3.2. Проверка на равенство

В котлине операторы равенства немного не такие как в джаве.

- `===` аналогично `==` в джаве.
- `!==` аналогично `!(a === b)`.
- `==` аналогично `a?.equals(b) ?: (b === null)` (равенство в обычном понимании).
- `!=` аналогично `!(a == b)`.

Операторы `===` и `!==` перегружать нельзя.

3.3.3. Конвенция invoke

Оператор `invoke()` – это что-то вроде `operator ()` В C++.

```
operator fun String.invoke(pref: String) {
    println(pref + this)
}

fun main(args: Array<String>) {
    "hello!"(">>> ") // напечатается ">> hello!"
}
```

3.3.4. Компоненты

Пусть есть `data`-класс, хранящий три значения, и мы хотим одним выражением присвоить их в три разные переменные.

```
data class Tripple<R, T, S>(val a: R, val b: T, val c: S)

fun foo(tripple: Tripple<Int, String, Char>) {
  val (x, y, z) = tripple // аналогично val x = tripple.a; val y = tripple.b; val z = tripple.c
}
```

То же самое можно делать и со списками.

```
fun bar(lst: List<String>) {
  val (first, second) = lst
}
```

Если в списке будет больше элементов, чем мы хотим получить, остальные элементы просто проигнорируются. Если же в списке элементов будет меньше, мы получим исключение `ArrayIndexOutOfBoundsException`.

Чтобы можно было деструктурировать класс на `N` компонент, для него надо перегрузить операторы `component1()`, ..., `componentN()`, каждая их которых возвращает соответствующий элемент. Для `data`-типов такие операторы перегружаются автоматически для всех полей. Для `List` определены только первые пять компонент, то есть по-умолчанию их нельзя разбивать более чем на пять компонент.

Для `Map.Entry` определены `component1()` и `component2()`. Поэтому по ним можно удобно итерироваться или вызывать `foreach`.

```
for ((key, value) in map) {
  println("$key: $value")
}
map.forEach { (key, value) -> println("$key: $value") }
```

3.3.5. Делегированные свойства

В джаве нам часто приходится писать много одинакового кода для работы с полями – лениво что-то вычислить, прочитать/записать, итд. В котлине, чтобы не писать кучу однотипных геттеров и сеттеров, придумали делегирование. Делегат – это класс, в котором определены операторы `getValue()` и `setValue()` со специальными сигнатурами.

```
class MyDeligate {
  operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
    // thisRef - ссылка на объект
    // property - ссылка на свойство этого объекта
  }

  operator fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {
    // thisRef - ссылка на объект, ссылка на свойство и новое значение
  }
}
```

```
class Example {
  var p: String by MyDeligate()

  fun foo() {
    p = ''asd'' // MyDeligate.setValue(''asd'')
    println(p) // println(MyDeligate.getValue())
  }
}
```

Запись `<*>` соответствует `<out Any>`, то есть любой тип.

3.4. Стандартные делегаты

3.4.1. `lazy`

`lazy` – это функция, которая принимает лямбду и возвращает делегат, который при первом обращении запускает эту лямбду, сохраняет результат и возвращает его при последующих обращениях.

```
val lazyValue: String = lazy {
  println(''computed!'') // вызовется один раз
  ''Hello'' // будет храниться
}
```

3.4.2. `Deligates.observable`

Это тоже функция, которая принимает начальное значение и лямбду и возвращает делегат, который при изменении переменной вызывает переданную лямбду. Например, можно залогировать изменение переменной.

```
import kotlin.properties.Delegates

var name: String by Delegates.observable(''init'') {
  prop, old, new -> println(''$old -> $new'') // prop - ссылка на переменную
}
```

3.5. `inline` функции

Такие функции будут целиком подставляться вместо вызовов. Это позволяет компилятору делать дополнительные оптимизации. Например, если `inline` функция принимает лямбду, то для нее не надо будет создавать отдельный класс.

На инлайн функции накладываются некоторые ограничения. Например, они не могу быть рекурсивными.

3.5.1. нелокальные возвраты

Есть у нас инлайн функция, принимающая лямбду. Тогда в теле этой лямбды мы можем делать `return` из нее, либо вообще из всех функций.

```
fun foo(lst: List<Int>) {
    lst.forEach {
      if (it == 42) return@forEach // выходим только из лямбды, что-то вроде continue
      if (it == 56) return // выходим из foo
    }
}
```

3.5.2. Reified параметры типов

В джававских дженериках происходит стирание типов, из-за чего нельзя пользоваться такими средствами языка, как, например, `(x instanceOf T)` или `x = getInstance(T.class)`. В котлине по-умолчанию типы тоже стираются, но для параметров инлайн функций это можно запретить ключевым словом `reified`. Тогда при подстановки тела функции, туда подставятся и нужные типы.

```
inline fun <reified T> Iterable<*>.filterIsInstance(): List<T> {
  val destination = arrayListOf<T>()
  for (element in this) {
    if (element is T) { // можем благодаря reified
      destination.add(element)
    }
  }
  return destination
}
```

3.6. Пример: Dependency injection

Рассмотрим пример, как разные фичи котлина взаимодействуют друг с другом и позволяют писать более красивый код.

Объявляем интерфес для контейнера и добавляем ему инлайн эктеншн оператор `getValue`, позволяющий делегировать нашему контейнеру получение инстанса произвольного дженерик типа (с помощью метода `getInstance()`). Инлайнинг и `reified` параметр позволяют нам получать класс дженерик типа (`T::class`). Теперь наш контейнер умеет выдавать значение нужного типа без использования громоздких конструкций.

```kotlin
interface Container {
  fun <T : Any> getInstance(klass: KClass<T>): T
}

inline operator fun <reified T : Any> Container.getValue(
    thisRef: Nothing?, kProperty: KProperty<*>
): T = getInstance(T::class)

fun doSomething(c: Container) {
  val myComp: MyClass by c
  myComp.hashCode() // равносильно c.getInstance(MyComp::class).hashCode()
}
```

3.7. Nullable операторы: `let` и `as?`

3.7.1. let

Пусть есть парсер, который возвращает либо распознанное значение, либо `null`. Если успешно распозналось, хотим сразу вернуть значение из функции.

```kotlin
fun foo() {
  val parser = ...
  val parsed = parser.parse()?.let { return it } // выходим из foo
  ... // действия в случае неудачного парсинга
}
```

3.7.2. as?

Хотим прикастовать один тип к другому, но не уверены, что можем.

```kotlin
fun foo(x: Any) {
  val y: Int = x as Int // если не Int, получим ClassCastException
  val z: Int = x as? Int // если не Int, вернется null
}
```

3.8. Еще немного анонимных функций

Поскольку анонимные функции можно присвоить в переменную, им можно приписать тип.

```kotlin
val usual = (String) -> Int = { x -> x.length }
val ext: String.() -> Int = fun String.(): Int = this.length
```

Экстеншн функции тоже могут быть анонимными.

```kotlin
val ext1 = fun String.(): Int = this.length // явно указывать тип не обязательно
val ext2: String.() -> Int = { this.length } // тип надо явно указать
val ext3: String.() -> Int = { length } // тип надо явно указать
```

3.9. Пример: функциональный литерал с ресивером

Функция `builder()` будет принимать функцию-расширение для `String` и применять ее к какой-то строке.

```
fun builder(block: String.() -> Int) = ''someString''.block()
fun String.foo() = 1

fun test() {
    // лямбда имеет тип String в качестве ресивера
    builder {
        this.length
        // равносильно
        this@builder.length
        // равносильно
        length
    }
    // передаем ссылку на функцию String.foo()
    builder(String::foo)
}
```

3.10. Стандартные функции высших порядков

3.10.1. `run`

В стандартной библиотеке котлина есть различные функции высших порядков, то есть работающие с функциями в качестве аргументов. Например, функция `run()` просто выполняет переданный ей блок кода.

```
inline fun <R> run(block: () -> R): R = block()
```

Такая функция позволяет, например, проинициализировать переменную какой-то сложной логикой, не вынося ее в отдельную функцию.

```
val x: Int = run {
  val result = findComponent() ?: return@run null
  val arg = findArgument() ?: return@run null
  result.callFunction(arg)
}
```

3.10.2. `let`

Эту функцию мы уже упоминали раньше, рассмотрим ее подробнее.

```
inline fun <T, R> T.let(block: (T) -> R): R = block(this)
```

В отличие от `run()`, функция `let()` является расширением для generic типа T, что позволяет применять переданный ей блок кода к результату какого-то выражения.

```
fun foo(): String? = ''foo''

val x = foo()?.let { x ->
  // some logic
  x.length
}
```

3.10.3. with

Функция `with()` похожа на `let()`, только она принимает функцию-расширение и применяет ее к ресиверу.

```
inline fun <T, R> with(receiver: T, block: T.() -> R): R = receiver.block()
```

Это бывает полезно, если надо вызвать подряд много методов у переменной или у результата какого-то выражения.

```
fun test(wordsToInts: MutableMap<String, Int>) {
  with(wordsToInts) {
    put(''one'', 1)
    put(''two'', 2)
  }
}
```

3.10.4. run с ресивером

Функция `run()` также реализована в качестве расширения.

```
inline fun <T, R> T.run(block: T.() -> R): R = block()
```

Теперь мы можем делать всё то же что и с `with()`, только еще предварительно проверять выражение на `null`.

```
val x = foo()?.run {
  // some logic
  length
} ?: 0
```

3.10.5. apply

Функция `apply()` выполняет переданный код над ресивером, а затем возвращает не то значение, которое вернул блок кода, а значение ресивера.

```
inline fun <T> T.apply(block: T.() -> Unit): T {
  block()
  return this
}
```

Теперь можно красиво заполнить список какими-то значениями и сделать его неизменяемым.

```
val lst = List<Int> = mutableListOf<Int>().apply {
  for (i in 1..10) {
    add(i * 2)
  }
}
```

Еще можно лаконично писать билдеры.

```
fun create(): Person {
  return Person().apply {
    name = ''Mike''
    age = 19
  }
}
```

3.10.6. `also`

Эта функция похожа на `apply()`, но принимает не расширение, а обычную функцию.

```
inline fun <T> T.also(block: (T) -> Unit): T {
  block(this)
  return this
}
```

Её можно применять, когда объект уже проинициализирован, и мы хотим его куда-то присвоить или вернуть из функции, но предварительно хотим что-то еще сделать (например, закешировать).

```
data class Meters(val m: String)
val cache = mutableMapOf<String, Meters>()
fun createOrGetCached(key: String): Meters = cache[key] ?: Meters(key).also { cache[key] = it }
```

4. Type-safe builders

С помощью `extension`-функций можно создавать прямо в синтаксисе котлина доменно специфичные языки (DSL). Такие языки можно использовать, чтобы генерировать разметку (HTML, XML, TeX), используя код на котлине. При этом будут работать различные проверки синтаксиса и типов, присущие данному языку.

Например, библиотека `kotlinx.html` позволяет генерировать HTML разметку следующим образом.

```
fun result(rows: Collection<Person>) =
  html {
   head {
     title { +''HTML encoding with Kotlin'' }
   }
   body {
     h1 { +''HTML encoding with Kotlin'' }
     for (person in rows) {
       div { +person.toString() }
     }
   }
  }
```

При этом мы хотим, например, чтобы тег `head` был доступен только внутри тега `html`. И чтобы компилятор поругался на нас, если мы случайно поместим `head` внутрь `body`.

Посмотрим, как это устроено внутри. Сделаем возможность создавать HTML объект, чтобы в дальнейшем его можно было как-то инициализировать.

```
class HTML

fun html(init: HTML.() -> Unit): HTML {
  val html = HTML()
  html.init()
  return html
}

fun test() {
   html {
      // this - член класса HTML
   }
}
```

Добавим в наш HTML элементы – Head и Body.

```
interface Element

class HTML : Element {
   class Head : Element
   class Body : Element

   val children = mutableListOf<Element>()

   fun head(init: Head.() -> Unit): Head {
     val head = Head()
     head.init()
     children.add(head)
     return head
   }
```

```
15
16    fun body(init: Body.() -> Unit): Body {
17      val body = Body()
18      body.init()
19      children.add(body)
20      return body
21    }
22  }
```

Теперь при вызове функций `head()` и `body()`, будут создаваться соответствующие элементы и добавляться в список детей HTML-объекта. Теперь можем писать следующий код.

```
1  fun test() {
2    html { // создали объект HTML
3      head { } // добавили в HTML голову
4      body { // добавили в HTML тело
5        head { } // всё еще можем внутри body написать head, так как здесь доступен ресивер HTML
6      }
7    }
8  }
```

Чтобы было удобно добавлять текстовые элементы, можно перегрузить унарный оператор `+` для строчек.

```
1  class TextElement(val text: String) : Element
2
3  operator fun String.unaryPlus() {
4    children.add(TextElement(this))
5  }
6
7  html {
8    body {
9      +''I am a text element''
10   }
11 }
```

Какие преимущества мы уже получили?

Во-первых, мы запрещаем добавлять элементы вне тегов, в которых они должны быть (например, `title` вне `head`).

Во-вторых, мы можем писать нетривиальный код создания и добавления элементов, а затем переиспользовать его, вынося в отдельные функции.

Однако, мы всё еще можем добавлять элементы на любом уровне вложенности (например, `head` внутри `body`).

```
1  html {
2    body {
3      head { } // this@html.head
4    }
5  }
```

То есть внутри блока `body` нам доступны два ресивера – `this` (относящийся к `Body`) и `this@html` (относящийся к `HTML`). Хотим разрешить использовать только ближайший ресивер. На уровне языка это сделать нельзя, чтобы не сломать логику в других местах, где нужны разные ресиверы.

Поэтому ввели специальную аннотацию `@DslMarker`, которая позволяет создавать свои аннотации, которые будут говорить компилятору, что помеченные ими классы являются элементами DSL, и в них, в частности, при неявном указании ресивера будет выбираться ближайший. Если нам всё же понадобился более внешний ресивер, его можно указать явно.

```
@DslMarker
annotation class HtmlTagMarker // объявили аннотацию-маркер

@HtmlTagMarker
interface Element // пометили интерфейс Element маркером

class HTML : Element { ... }

html {
   body {
      head { } // ошибка, неявный ресивер - Body
      this@html.head { } // ок - явно указали ресивер
   }
}
```

5. Взаимодействие с Java

В котлин изначально заложена двусторонняя совместимость с Java. Для этого создателям котлина пришлось приложить определенные усилия, например, чтобы поддержать корректную компиляцию проектов, в которых присутствует код сразу на двух языках. Компилятор котлина умеет читать java-файлы и извлекать из них необходимую для взаимодействия информацию. Компилятор java, в свою очередь, ничего не знает про котлин. Поэтому при компиляции смешанного проекта, сначала запускается компилятор `kotlinc`, который генерирует class-файлы, и результат передается компилятору `javac` в виде байт-кода, понятного java-машине.

Если взять простой код на джаве, переписать его на котлин и скомпилировать, то получится примерно такой же байт-код, как и при компиляции джавы. Однако, у котлина есть особые фичи, для которых надо придумывать различные костыли для совместимости.

Например, для свойств в классах создаются приватные поля, геттеры и сеттеры, а обращение к этим свойствам заменяется на вызов соответствующих геттеров и сеттеров.

5.1. Классы для файлов

В котлине свойства и функции можно объявлять на уровне файла. В джаве так нельзя. Поэтому для каждого такого файла сгенерируется класс `FileNameKt`, к которому потом можно будет обращаться из джава-кода.

```
// file: example.kt
val property = 42
fun foo() = ''Hello''

//Java:
String s = ExampleKt.foo();
int x = ExampleKt.getProperty();
```

Если хочется чтобы для файла сгенерировался класс с кастомным именем, в начале файла можно написать специальную аннотацию `@file:JvmName`.

```
// file: example.kt
@file:JvmName(''Utils'')
fun foo() = ''Hello''

//Java:
Utils.foo()
```

Также есть аннотация `@file:JvmMultifileClass`, чтобы для нескольких файлов сгенерировался один общий класс.

```
// file: example.kt
@file:JvmName(''Utils'')
@file:JvmMultifileClass
fun foo() = ''Hello''

// file: anotherExample.kt
@file:JvmName(''Utils'')
@file:JvmMultifileClass
fun bar() = ''World''
```

5.2. Объекты – синглтоны

Объекты и `companion`-объекты компилируются в singleton-классы, и получать доступ к ним надо через специальные поля. При этом все члены singleton-класса будут не статическими, а принадлежащими единственному экземпляру.

```
object MyObj {
   fun foo() { }
}

class MyClass {
   companion object {
      fun bar() { }
   }
}

// Java:
MyObj.INSTANCE.foo();
MyClass.Companion.bar();
```

5.3. Статические свойства и методы

Но иногда нам всё же нужны настоящие статические свойства и методы. Например, когда мы получили объект какого-то класса из джавы, в котором они есть. Чтобы объявить настоящий статический член класса в котлине, надо объявить его в объекте или в companion-объекте и добавить ему аннотацию `@JvmStatic`.

```
object MyObj {
  @JvmStatic
  fun foo() { }
}

class MyClass {
  companion object {
    @JvmStatic
    fun bar() { }
  }
}

// Java:
MyObj.foo();
MyClass.bar();
```

5.4. JVM-поля

Чтобы вместо свойства сгенерировать настоящее jvm-поле, можно использовать аннотацию `@JvmField`.

```
class MyClass {
  @JvmField
  val x = 42
}

// Java:
MyClass mc = new MyClass();
int x = mc.x;
```

5.5. Модификатор `const`

Еще в котлине есть модификатор `const`, позволяющий задать константное выражение. Это выражение, значение которого известно на этапе компиляции. Такие выражения можно использовать в java-аннотациях.

5.6. Стирание типов

В котлине мы можем объявить методы-расширения для типизированных списков двух разных типов.

```
fun List<String>.filterValid(): List<String>
fun List<Int>.filterValid(): List<Int>
```

Так как в джаве generic-типы стираются, такие две функции получат одну и ту же сигнатуру, и наш код на котлине не скомпилируется.

Чтобы такой код всё же скомпилировать, надо с помощью аннотации `@JvmName` указать другое имя для функции, которое будет в байт-коде. При этом из котлина можно будет вызывать метод со старым названием.

```
fun List<String>.filterValid(): List<String>
@JvmName(''filterValidInt'')
fun List<Int>.filterValid(): List<Int>

// Kotlin:
mutableListOf(1).filterValid()
mutableListOf(''1'').filterValid()

// Java:
filterValid(mutableListOf(''1''));
filterValidInt(mutableListOf(1));
```

5.7. Параметры по умолчанию

В котлине можно задавать значения по умолчанию для параметров функций.

```
fun foo(x: Int = 1, y: String = '''') { }
```

В джаве так нельзя, поэтому в байт-коде для таких функций сгенерируется обычный метод, а также метод со всеми аргументами и маской, говорящий, для скольки последних аргументов надо использовать значения по умолчанию. Это сделано для того, чтобы избежать генерации кучи перегруженных функций для всех вариантов.

```
void foo(int x, String y) { }
void foo$default(int x, String y, int nonDefaultsMask) { }
```

Если всё-таки хочется сгенерировать все перегрузки, можно написать аннотацию `@JvmOverloads`.

```
@JvmOverloads
fun foo(a: String, b: Int = 0, c: String = ''abc'') { ... }

void foo(String a) { ... }
void foo(String a, int b) { ... }
void foo(String a, int b, int c) { ... }
```

5.8. Проверяемые исключения

В котлине нет проверяемых исключений (тех, которые обязательно надо ловить). Это значит, что компилятор не будет заставлять нас оборачивать вызов функции в try-catch. В том числе и при вызове функции из джавы. Чтобы в джаве компилятор всё же ругался, надо приписать функции аннотацию `@Throws`

```
// Kotlin:
fun foo() {
   throw IOException()
}

@Throws(IOException::class)
fun bar() {
   throw IOException()
}

//Java:
foo(); // OK, скомпилируется
bar(); // Не скомпилируется - надо обернуть в try-catch
```

5.9. Declaration-site variance

В котлине есть (ко-/контр-)вариантность (см выше).

```
class Box<out T>(val value: T)

interface Base
class Derived : Base

fun boxDerived(value: Derived): Box<Derived> = Box(value)
fun unboxBase(box: Box<Base>): Base = box.value

val derived = Derived()
val alsoDerived = unboxBase(boxDerived(derived))
```

В джаве такого нет, поэтому данные функции будут транслироваться нетривиальным образом.

```
Box<Derived> boxDerived(Derived value) { ... }
Base unboxBase(Box<? extends Base>) { ... }
Base unboxString(Box<String>) { ... } // String - final класс, ? extends не нужен
```

Если ковариантный класс используется в качестве параметра функции, то он перейдет в `<? extends T>`. Если в качестве возвращаемого значения – то будет явный тип, без вайлдкардов, потому что иначе мы бы не смогли вызывать некоторые методы (если тип финальный, то вайлдкард не добавится).

5.10. Nullable типы

Возьмем обычную функцию на джаве и попытаемся вызвать её из котлина.

```
String foo(String x) { ... }
```

Какая сигнатура у этой функции будет в котлине? Поскольку в джаве все ссылочные типы nullable, было бы неправильно трактовать их в котлине как notnull. Однако, если все типы из

джавы трактовать как nullable (`String?, MyClass?, ...`), то код превратится в кучу вопросиков, и ненужных проверок на то, что переменные на самом деле не `null`. В итоге решили ввести так называемые платформенные типы `T!` для взаимодействия с джавой.

Такие типы обладают рядом свойств.

- `T! = T..T?`
- `T? <: T!`
- `T! <: T`
- `T <: T?, T? <: T! ⇒ T <: T!`
- `T! <: T, T <: T? ⇒ T! <: T?`
- `X = Y ⇔ X <: Y, Y <: X`
- `T? = T! = T`
- `T? != T` (нетранзитивное равенство!!!)

То есть получили сломанную систему типов, различные проблемы в компиляторе, зато можно писать красивый и более-менее типо-безопасный код.

Чтобы избежать платформенных типов, придумывают различные аннотации, например, знакомые нам из курса джавы аннотации `@Nullable` и `@NotNull` из `org.jetbrains.annotations`. Они подсказывают компилятору котлина, что в проаннотированных местах должны быть типы `T?` или `T`.

Платформенные типы также есть для mutable/immutable коллекций, ковариантных массивов и RAW типов. Они работают примерно так же и имеют те же проблемы что и платформенные типы для nullable/notnull.

5.11. Функциональные интерфейсы

В джаве есть функциональные интерфейсы. Это интерфейсы, у которых ровно один абстрактный метод, и для таких интерфейсов сделана возможность писать имплементацию в виде лямбды.

```
interface Runnable {
   void run();
}

Runnable r = () -> { System.out.println("Hello world!") };
r.run();
```

В котлине такие лямбды полностью заменяются концепцией функциональных типов. При этом для совместимости в котлин добавлена поддержка таких лямбд тоже.

```
interface Runnable {
    fun run()
}

val r = Runnable { println("Hello world!") }
```

Геттеры и сеттеры

Поскольку в котлине принята конвенция о том чтобы не писать геттеры и сеттеры, любой метод, начинающийся на `'get'`, можно вызвать из котлина просто как обращение к свойству.

```
// Java:
class Calendar {
   public int getFirstDayOfWeek() { ... }
}

// Kotlin:
val calendar = Calendar()
val day = calendar.firstDayOfWeek
```

6. Coroutines

6.1. Sequence

В Java 8 есть `stream`'ы, которые позволяют делать ленивые вычисления и создавать бесконечные последовательности. В котлине для этого есть интерфейс `Sequence` и различные функции для работы с ним.

Чтобы создать, например, бесконечную последовательность натуральных степеней двойки, достаточно вызвать функцию `generateSequence()` и передать в неё начальное значение и функцию получения следующего элемента из предыдущего.

```
val powerOfTwo = generateSequence(seed = 1) { 2 * it }
println(powerOfTwo.take(5).joinToString(" ")) // 1 2 4 8 16
```

Рассмотрим как Sequence работает изнутри. Интерфейс `Sequence` имеет метод `iterator()`, который возвращает обычный итератор, с помощью которого работают такие функции как `take()`, `map()`, `filter()`, и.т.п.

```
interface Sequence<out T> {
  operator fun iterator(): Iterator<T>
}
```

Функция `generateSequence()` возвращает экземпляр класса `GeneratorSequence`, который реализует интерфейс `Sequence`.

```
fun <T : Any> generateSequence(seed: T?, nextFunction: (T) -> T?): Sequence<T> =
  GeneratorSequence(seed, nextFunction)
```

Осталось понять, как класс `GeneratorSequence` реализует реализует итератор.

```
class GeneratorSequence<T : Any>(val seed: T?, val nextValue: (T) -> T?) : Sequence<T> {
  inner class SequenceIterator : Iterator<T> {
    var nextItem: T? = null

    // -2 for initial unknown
    // -1 for next unknown
    // 0 for done
    // 1 for continue
    var nextState: Int = -2
    ...
  }

  override operator fun iterator(): Iterator<T> = SequenceIterator()
}
```

У итератора есть несколько состояний.

-2 начальное состояние, когда еще ничего не проинициализировано;

-1 следующий элемент еще неизвестен;

0 следующий элемент – `null`, это значит, что последовательность закончилась;

1 можно продолжать итерироваться.

Вычислением следующего элемента занимается метод `calcNext()`. Если находимся в начальном состоянии, то инициализируемся переданным значением, иначе вызываем переданную функцию от текущего значения. Если текущее значение стало `null`, переходим в конечное состояние.

```
private fun calcNext() {
  nextItem = if (nextState == -2) getInitialValue() else getNextValue(nextItem!!)
  nextState = if (nextItem == null) 0 else 1
}
```

Метод `next()` проверяет, что мы не в конечном состоянии, получает, если нужно, следующий элемент и возвращает его, если тот не `null`.

```
override fun next(): T {
  if (nextState < 0) calcNext()

  if (nextState == 0) throw NoSuchElementException()

  val result = nextItem as T
  nextState = -1
  return result
}
```

Метод `hasNext()` получает, если нужно, следующий элемент и проверяет, что он действительно есть.

```
override fun hasNext(): Boolean {
  if (nextState < 0) calcNext()
  return nextState == 1
}
```

Функции для последовательностей – `map()`, `filter()` и им подобные, должны работать лениво, поэтому они просто оборачивают переданные им последовательности в специальные декораторы, добавляющие итераторам нужную логику.

```
public fun <T, R> Sequence<T>.map(transform: (T) -> R): Sequence<R> {
  return TransformingSequence(this, transform)
}
```

6.2. Sequence + yield

Усложним пример. Теперь у нас есть бесконечный цикл, активно генерирующий значения, мы хотим превратить его в ленивый.

```
fun fibonacci() = sequence {
    var a = 0
    var b = 1
    var temp: Int
    while (true) {
        yield(a) // магия, возвращающая значение и продолжающая вычисления
        temp = a
        a = b
        b += temp
    }
}

println(fibonacci().take(8).toList()) // 0, 1, 1, 2, 3, 5, 8, 13
```

Функция `sequence()` принимает экстеншн функцию для класса `SequenceScope`, в котором, собственно, и объявлен магический метод `yield()`. Также видим везде магическое слово `suspend`.

```
public fun <T> sequence(block: suspend SequenceScope<T>.() -> Unit): Sequence<T>

public abstract class SequenceScope<in T>() {
  public abstract suspend fun yield(value: T)
  ...
}
```

Когда лямбда передается с типом `suspend`, она трансформируется в специальный класс. Посмотрим на него. В нем есть локальные переменные из нашей лямбды, а так же текущее состояние. Вся логика находится в методе `resume()`. Он проверяет, в каком состоянии находится, и в зависимости от этого выполняет разные части лямбды.

```
class <anonymous_for_state_machine>
extends CoroutineImpl<...> implements Continuation<Object> {
  // Текущее состояние
  int label = 0;

  // локальные переменные из лямбды
  Integer a = null;
  Integer b = null;
  Integer temp = null;

  void resume(Object data) {
    if (label == 0) goto L0;
    if (label == 1) goto L1;
    else throw IllegalStateException();
    ...
  }
}
```

В нашем случае с `fibonacci()` у `label` будет два состояния, а тело лямбды преобразуется следующим образом.

```
L0:
  a = 0
  b = 1
  temp = 0
  label = 1
LOOP:
  label = 1
  data = yield(a, this)
  if (data == COROUTINE_SUSPENDED) return
L1:
  temp = a;
  a = b;
  b += temp;
  label = -1
  goto LOOP
END:
  label = -1
  return
```

Изначально `label == 0`, поэтому код начнется с начала, войдет в цикл и вызовет метод `yield()`. После этого проверит, не прервали ли нас и, если прервали, то выйдет. При следующем вызове `resume()` у нас будет состояние `label == 1`, поэтому мы пойдем сразу на метку `L1` и продолжим выполнение цикла, пока снова не попадем на `yield()`.

Таким образом, если мы вызовем функцию `fibonacci()`, она нам вернет `Sequence`. У него мы возьмем итератор, и при каждом вызове `next()`, будет вызываться сгенерированный метод `resume()`, доходить до `yield()`, возвращать следующее значение и прерываться до следующего вызова `next()`.

Сгенерированный для лямбды класс называется корутина.

Реализация итератора для таких последовательностей похожа на то, что мы видели для `GeneratorSequence`, только, помимо прочего, он содержит метод `yield()`, который вызывается из нашей корутины. Он сохраняет переданное значение в качестве следующего элемента, переходит в состояние `State_Ready` и возвращает маркер `COROUTINE_SUSPENDED`, который говорит корутине, что ей надо прерваться.

```
suspend override fun yield(value: T) {
  nextValue = value
  state = State_Ready
  return suspendCoroutineOrReturn { c ->
    nextStep = c
    COROUTINE_SUSPENDED
  }
}
```

6.3. Continuation

В рассмотренном примере контроль управления передается явно через продолжения (Continuation). Этот подход похож на коллбеки и заключается в следующем – есть интерфейс `Continuation` с методом `resume()` и сложная функция, которая что-то вычисляет. Это всё передается в код, который запускает сложную функцию, дожидается её завершения, после чего вызывает `Continuation.resume()`, который далее что-то делает с результатом сложной функции.

```
interface Continuation<T> {
  fun resume(t: T)
}

fun <T> yield(complexFun: () -> T, c: Continuation<T>) {
    val value = complexFun()
    c.resume(value)
}
```

Рассмотрим пример использования такого подхода. Передадим в функцию `yield()` лямбду, которая просто вызывает `foo()`, а в качестве коллбека – `Continuation`, внутри которого еще раз вызовем `yield()` с лямбдой, вызывающей `bar()` и еще одним `Continuation`, который вызывает `baz()`.

```
yield({ foo() }, object : Continuation<Int> {
  override fun resume(t1: Int) {
    // Здесь значение уже точно вычислено
    yield({ bar(t1) }, object : Continuation<Int> {
      override fun resume(t2: Int) { baz(t2) }
    })
  }
})
```

Таким образом мы построили цепочку из двух коллбеков, которые будут нам гарантировать последовательное выполнение `t1 = foo()`, `t2 = bar(t1)` и `baz(t2)`.

С увеличением вложенности, растет количество скобочек и кода в целом, при этом сильно

снижается читаемость кода. Такая проблема называется Callback Hell, она распространена в JavaScript и других похожих языках.

Чтобы избежать коллбек-ада, в котлине введено ключевое слово `suspend`, которое при трансляции в байт-код берет функцию и разворачивает её в корутину, которая уже работает на коллбеках и `yield`'ах.

6.4. Suspend-функции

На `suspend`-функции накладываются ограничения. Так как к таким функциям при компиляции добавляется еще один параметр, который берется из сгенерированного кода, `suspend`-функции можно вызывать только из других `suspend`-функций или из `suspend`-лямбд.

```
suspend fun foo() { ... }

fun usual() { foo() } // Ошибка

suspend fun other() { foo() } // OK

fun func(f: suspend () -> Unit) { ... }
func { foo() } // OK
```

Такой код будет превращаться в следующее.

```
suspend fun foo() { ... }
suspend fun other() { foo() }
// превратится в
fun foo(c: Continuation): Any?
fun other(c: Continuation): Any { foo(c) }
```

То есть при вложенных вызовах, Continuation будет передаваться во все функции.

6.5. Разоблачение магии

Разберемся, как же работает `SequenceBuilderIterator`. Для этого вернемся к методу `yield()`. Он возвращает такую страшную конструкцию.

```
return suspendCoroutineUninterceptedOrReturn { c ->
  nextStep = c
  COROUTINE_SUSPENDED
}
```

Suspend-функция `suspendCoroutineUninterceptedOrReturn()` – это заглушка, позволяющая вытащить переданный ей неявно `Continuation`, чтобы в `yield()` его можно было сохранить в итераторе и вызвать на следующем шаге.

Итак, что же происходит с нашей функцией вычисления фибоначчи при компиляции. Лямбда, переданная в функцию `sequence()`, превращается в анонимный класс, унаследованный от `Continuation`. Тело лямбды разбивается на куски между вызовами `yield()` и попадает в метод `resume()`. Каждый вызов `yield(a)` заменяется на вызов `yield(a, this)`, где `this` – это тот же самый объект, который сейчас работает. Таким образом, наша лямбда сказала итератору, чтобы на следующем шаге он вызвал её еще раз. Поэтому, при следующем вызове `iterator.next()`, снова вызовется наша бывшая лямбда, только уже в новом состоянии, и выполнит еще один шаг вычислений.

7. Асинхронное программирование

В современных программах большая часть кода чего-то ждет – получения данных, пользовательского ввода, итд. Рассмотрим простой пример. У нас есть UI, который просит пользователя ввести некий `token`, затем по этому токену и некоторому объекту `item` делаем запрос на сервер и как-то обрабатываем ответ.

```
fun postItem(item: Item) {
  val token = requestToken()
  val post = createPost(token, item)
  processPost(post)
}
```

7.1. Потоки и блокировки

Одна из возможных реализаций функции `requestToken()` – это ручная блокировка, что-то типа спинлока. Показываем пользователю UI с формой и в цикле ждем, пока он не заполнит форму.

```
fun requestToken(): Token {
  // показать UI сформой
  ...

  while (true) {
    if (ready()) {
      return token
    }
  }
}
```

Активное ожидание – это не всегда хорошо, поэтому можно вынести UI в отдельный поток и заджойнить его.

```
fun requestToken(): Token {
  val t = Thread {
    // показать UI с формой в отдельном потоке
  }
  t.start()

  // заблокироваться на поток, пока не получим результат
  t.join()
  return token
}
```

Однако плодить потоки мы тоже не всегда хотим. В частности, каждый поток отбирает порядка 2мб дополнительной памяти. А в некоторых языках, например в JavaScript, потоков нет совсем.

7.2. Callbacks

Следующий вариант – колбеки. Передаем в функцию `requestTokenAsync()` лямбду, которая выполнится, когда токен будет получен.

```
fun requestTokenAsync(callback: (Token) -> Unit) { ... }
```

Тогда наша основная функция будет выглядеть так.

```
fun postItem(item: Item) {
    requestTokenAsync { token ->
        createPostAsync(token, item) { post ->
            processPost(post)
        }
    }
}
```

Видим кучу вложенных лямбд и вспоминаем про Callback hell. Чтобы этого избежать, в других языках есть концепции Future/Promises/Rx, это объекты, которые возвращаются из функций с асинхронной логикой и инкапсулируют в себе колбеки, позволяя писать более простой и понятный код.

```
fun requestTokenAsync(): CompletableFuture<Token> { ... }

fun postItem(item: Item) {
    requestTokenAsync()
        .thenCompose { token -> createPostAsync(token, item) }
        .thenAccept { post -> processPost(post) }
}
```

Метод `thenCompose()` дожидается результата `CompletableFuture` и вызывает переданный ему колбек. Метод `thenAccept()` разворачивает все вложенные друг в друга `CompletableFuture`, дожидаясь их всех, и обрабатывает результат. Методов для работы с `CompletableFuture` много, их все приходится помнить, а работать с ними сложнее чем с обычным последовательным кодом.

7.3. Корутины

Котлин предлагает альтернативный подход – корутины. Добавим к сигнатурам наших асинхронных функций волшебное слово `suspend`. Главную функцию тоже сделаем `suspend`. Теперь можем писать обычный последовательный код.

```
suspend fun requestTokenAsync(): Token { ... }
suspend fun createPostAsync(token: Token, item: Item): Post { ... }

suspend fun postItem(item: Item) {
  val token = requestTokenAsync()
  val post = createPostAsync(token, item)
  processPost(post)
}
```

Посмотрим, во что компилятор трансформирует такую функцию. Ей в параметры добавляется объект типа `Continuation`. Затем создается конечный автомат, у которого есть метод `resumeWith()`. Весь код функции разбивается на блоки между `suspend`-вызовами. В каждом блоке выполняется логика и вызывается вложенная `suspend` функция, при этом автомат передается в неё. Таким образом, `Continuation` в данном случае является подобием колбека, который вызывает эту же функцию, с новым состоянием автомата.

```
fun postItem(item: Item, c: Continuation) {
  val stateMachine = c ?: object : Continuation<Unit> {
    override fun resumeWith(result: Result<Unit>) {
      postItem(null, this)
    }
  }

  when (stateMachine.label) {
    0 -> {
      stateMachine.item = item
      stateMachine.label = 1
      requestTokenAsync(stateMachine)
    }
    1 -> {
      createPostAsync(token, item, stateMachine)
    }
    ...
  }
}
```

Но, поскольку функция `postItem()` теперь `suspend`, её нельзя вызвать из нормальной функции. Вместо этого можно использовать coroutine builder. Для этого просто передадим наш код в функцию `launch()` из библиотеки `kotlinx.coroutines`. Она выполнит код в фоновом потоке.

```
fun postItem(item: Item) {
  runBlocking {
    val job = launch {
      val token = requestTokenAsync()
      val post = createPostAsync(token, item)
      processPost(post)
    }
    job.join()
  }
}
```

Важно заметить, что весь код, в том числе и обработка результата (`processPost()`) выполнится в каком-то другом потоке, но иногда мы хотим что-то обрабатывать в конкретном потоке. Например, в android трогать UI можно только из UI-потока. Поток можно задать в параметре `launch` с помощью `Dispatcher`.

```
launch(Dispatchers.Main) { ... }
```

7.4. Экскурс в C# – async/await

В языке C# с асинхронными функциями можно работать через механизм async/await. Функция, помеченная ключевым словом async, возвращает таску, в которой будет лежать результат, когда будет готов. Если же перед вызовом такого метода написать слово await, то функция выполнится синхронно, то есть мы дождемся её завершения и получим готовый результат.

```
// C#
async Task<Token> requestToken() { ... }
async Task<Post> createPost(Token token, Item item) { ... }
void processPost(Post post) { ... }

async task postItem(Item item) {
  var token = await requestToken();
  var post = await createPost();
  processPost();
}
```

Если же вызвать async функцию без слова await, то она начнет выполняться в другом потоке, а нам сразу вернется объект типа Task, с помощью которого мы позже сможем так же дождаться результата. Такой подход сохраняет параллелизм. Убедимся в этом на примере с параллельной загрузкой двух картинок.

```
// C#
async Task<Image> loadImage(String name) { ... }

var task1 = loadImage(''img1'');
var task2 = loadImage(''img2'');
// картинки уже начали параллельно загружаться
var image1 = await task1;
var image2 = await task2;
```

7.5. async/await в котлине

В котлине async – это не ключевое слово, а просто функция, принимающая на вход корутину и возвращающая объект типа Deferred (аналог Task). Эта функция находится в библиотеке kotlinx.coroutines. Ключевого слова await в котлине тоже нет, зато у Deffered есть метод await(), который и содержит логику ожидания и получения результата. В итоге, на котлине код из предыдущего примера будет выглядеть так.

```
// Kotlin
fun loadImage(name: String): Deferred<Image> = async { ... }

val deferred1 = loadImage(''img1'')
val deferred2 = loadImage(''img2'')

val image1 = deferred1.await()
val image2 = deferred2.await()
```

Более явный способ использования `async/await` – сделать `loadImage()` обычной функцией, а её вызов обернуть в `async{ ... }`.

```
// Kotlin
fun loadImage(name: String): Image { ... }

val deferred1 = async { loadImage(''img1'') }
val deferred2 = async { loadImage(''img2'') }

val image1 = deferred1.await()
val image2 = deferred2.await()
```

В таком случае мы сможем, когда надо, запускать функцию асинхронно, а обычный вызов будет последовательным. Таким образом мы подчеркиваем, что по-умолчанию у нас всё последовательно, а чтобы запустить что-то асинхронно, надо об этом явно заявить.

7.6. Ментальная модель корутин

О корутинах можно думать, как о легковесных потоках, ведь они занимают гораздо меньше дополнительных ресурсов, чем настоящие java-потоки. Например, мы можем создать сто тысяч корутин и без проблем их запустить.

```
fun foo() {
  runBlocking {
    val jobs = List(100_000) {
      launch {
        delay(1000)
        println(''...'')
      }
    }

    jobs.forEach { it.join() }
  }
}
```

Аналогичный же код с настоящими java-потоками на большинстве компьютеров упадет с ошибкой `OutOfMemoryError`.

7.7. Шаблон CSP

Communicating Sequential Processes – это шаблон проектирования распределенных систем. До того как этот шаблон придумали, для взаимодействия между потоками использовался shared state, что вело к куче проблем и сложностей с синхронизацией и эффективностью. Шаблон CSP предлагает вместо общего состояния использовать альтернативные способы общения между потоками.

7.7.1. Channel

Один из способов межпроцессного взаимодействия – это каналы. Запустим параллельно две корутины, одна будет писать числа в канал, а другая читать числа из канала и выводить в stdout.

```
fun foo() = runBlocking<Unit> {
    val channel = Channel<Int>()

    launch(coroutineContext) {
        repeat(10) {
            delay(100)
            channel.send(it)
        }
        channel.close()
    }

    launch(coroutineContext) {
        for (x in channel) {
            println(x)
        }
    }
}
```

7.7.2. Actors

В классическом CSP мы явно работаем с объектами, через которые происходит взаимодействие (например с каналами). В модели акторов мы делаем по-другому. Актор – это именованная корутина и канал, с помощью которого она сможет общаться с другими акторами. Когда один актор хочет что-то сообщить другому, он вызывает у того соответствующий метод, в котором сокрыта логика работы с каналами.

```
runBlocking<Unit> {
    val printer = actor<Int>(coroutineContext) {
        for (i in channel) {
            println(i)
        }
    }

    launch(coroutineContext) {
        repeat(10) {
            delay(100)
            printer.send(it)
        }
        printer.close()
    }
}
```

Посмотрим, как можно реализовать подсчет чисел фибоначчи с помощью акторов.

```
runBlocking {
    val fibonacci = produce { // создаем актора
        var a = 1
        var b = 1
        while (true) {
            send(a)
            val temp = a + b
            a = b
            b = temp
        }
    }

    println(fibonacci.receive()) // 1
    println(fibonacci.receive()) // 1
    println(fibonacci.receive()) // 2

    fibonacci.cancel() // прерываем актора
}
```

Функция `produce()` создает актора, а внутри мы вместо `yield()` используем `send()` для отправки следующего числа наружу. Важно, что функция `send()` засыпает до тех пор, пока отправленное значение не прочитают. Поэтому, после того как мы прочитали столько значений, сколько нам нужно, мы прерываем актора методом `cancel()`.

7.8. Язык vs библиотека

Современные языки стараются делать как можно меньше. Это проще для переносимости и для совместимости с разными платформами. Поэтому в котлине для корутин на уровне языка реализованы только ключевое слово `suspend` и возможность трансформировать корутину в конечный автомат. Всё остальное – `async/await`, каналы, акторы и прочее – реализовано в виде библиотеки, написанной уже на чистом котлине. Про все возможности библиотеки `kotlinx.coroutines` можно прочитать здесь.