Язык Kotlin: базовые конструкции

Структура программы

Текст программы на языке Kotlin состоит из нескольких основных блоков:

```
package имя.пакета

import пакет.1
import пакет.2

fun main() {
    код программы
}
```

Первая строка в файле может содержать ключевое слово `package` и имя пакета. Это позволяет сгруппировать схожие по смыслу классы и функции в одном пакете, а затем по необходимости импортировать их в другие файлы с кодом. Например, пакет `com.myproject.calc` может содержать вспомогательные вычислительные функции, пакет `com.myproject.graphics` – функции для рисования и т. д.

Имя пакета обычно формируется как имя домена в интернете, только в обратном порядке. Например, если сайт в интернете имеет имя `myproject.mycompany.com`, то пакет можно назвать `com.mycompany.myproject`. Это позволит избежать одинаковых названий с другими пакетами. Если своего домена нет, то название можно выбирать любое, главное, чтобы оно было в достаточной степени уникальным.

Использование имен пакетов также позволяет избежать конфликта имен функций и классов. Например, имя `Line` может использоваться для графического класса, рисующего линию на экране, а может для геометрического класса, вычисляющего математические параметры. Если разнести эти классы по своим пакетам, то конфликта не будет.

В случае написания программы в среде Kotlin Playground строку с директивой package можно не писать.

Следующие строки с директивами `import` позволяют импортировать классы и функции из других пакетов. В простых программах, не использующих сторонние пакеты, строки с директивами import также можно не включать.

Наконец, функция `main` – это главная точка входа в программу. Именно функция `main` будет выполняться первой при запуске консольной программы, поэтому её наличие является обязательной. Однако в других областях (например, при написании программы для ОС Android) точки входа другие, там функции `main` быть не должно.

Комментарии

Помимо кода программа может содержать пояснения для людей, читающих текст программы. Хорошим тоном будет пояснять отдельные места кода, которые не очень просты для понимания. Даже если при написании программы их назначение кажется очевидным, через какое-то время они могут вызывать недоумение даже у их автора.

Комментарии в Kotlin, как и во многих других языках программирования, могут быть однострочными и многострочными.

Однострочные комментарии начинаются со знака двух слешей (`//`) и продолжаются до конца строки:

```
// Это комментарий
println("Привет!") // И это комментарий
```

Многострочные комментарии располагаются между знаками `/*` и `*/`.

```
println("Привет!") /* Это
многострочный
комментарий */
println("Пока!")
```

Многострочный комментарий на самом деле может располагаться и в одной строке, например, когда нужно временно отключать какую-то часть кода или пояснить её:

```
val n = Math.sin(x) + /* Math.cos(x) + */ Math.tan(x)
```

В этом примере закомментировано слагаемое с функцией косинуса, в вычислениях оно не будет использоваться.

Даже если в тексте комментария встречаются операторы или директивы языка – они тоже будут игнорироваться при компиляции программы.

Переменные

Переменная – это ячейка памяти, в которой может храниться некоторое значение.

Чтобы к такой ячейке было удобнее обращаться, ей даётся имя. В языке Kotlin имя переменной может состоять из буквенных символов, цифр и знака подчёркивания, но не может начинаться с цифры. Примеры допустимых имен переменных:

- `i`
- `myNumber`
- `my_number`
- `_number`
- `number_12`
- `число`
- `моя_переменная`

Регистр символов в имени переменной имеет значение: переменные `my_number` и `My_number` – это разные переменные, и если переменная объявлена как `my_number`, а в другом месте к ней попытаться обратиться по имени `My_number`, то при компиляции программы будут показаны ошибки.

Перед тем как использовать переменную, её нужно объявить, то есть описать как она будет называться, какой тип данных в ней может храниться. Для описания переменных используются ключевые слова `var` и `val`.

Переменную, описанную с ключевым словом `var`, можно изменять сколько угодно раз:

```
var n = 5
n = 7
n = 8
```

А вот переменную, описанную с ключевым словом `val`, можно присвоить значение только один раз:

```
val k = 10
k = 15 // Здесь возникнет ошибка!
```

Такое «одноразовое» присваивание может быть полезно, это позволяет избежать случайного изменения значения, которое должно оставаться постоянным.

У каждой переменной есть своя **область видимости** – блок программы, в котором к этой переменной можно обращаться. За пределами блока видимости переменная не видна, и при попытке обратиться к ней будет показана ошибка. Например, если в программе есть две функции, то в каждой из них может быть своя переменная с именем `m`:

```
fun main() {
    var m = 5
}

fun calc() {
    var m = 7
}
```

Но в рамках одного блока видимости описать переменную второй раз уже нельзя:

```
fun main() {
    var m = 5
    var m = 7 // Ошибка!
}
```

Типы данных

Помимо задания имени переменной, нужно определить ещё тип данных, которые переменная сможет хранить.

Kotlin – это статически типизируемый язык, то есть тип переменной задаётся на этапе написания и компиляции программы, и уже не может быть изменён в процессе её выполнения. Это означает, что если переменная заявлена для хранения целых чисел, в неё уже не получится записать число с плавающей точкой или текстовую строку.

Типы данных можно разделить на несколько категорий:

- Целые числа: `Byte`, `Short`, `Int`, `Long`
- Числа с плавающей точкой: `Float`, `Double`
- Символы и строки: `Char`, `String`
- Логический тип: `Boolean`

Когда переменная объявляется, компилятор пытается определить тип данных, который будет за ней закреплён. Если вместе с объявлением переменной присваивается какое-то значение (производится её *инициализация*), то компилятор может определить тип данных по значению:

```
val number = 5
val text = "Привет"
```

Так как переменной `number` присваивается целое число, её тип будет определен как `Int`. В свою очередь, переменной `text` присваивается строка, поэтому её тип будет `String`.

Но можно указать тип переменной и явно, написав его после двоеточия:

```
val number: Int = 5
val text: String = "Привет"
```

Если же значение переменной планируется присвоить позже, а не в момент объявления, то тогда указывать тип нужно обязательно, компилятор не сможет определить его самостоятельно:

```
var number: Int
...
number = 8
```

В отличие от ряда других языков, даже базовые типы (такие как числа или строки) представляют собой полноценные объекты. Это значит, что над ними можно легко выполнять различные дополнительные операции. Так, например, чтобы преобразовать число в строку, нужно вызвать метод `toString()`:

```
val number = 5
val text = number.toString()
```

Подробнее о встроенных функциях будет рассказано далее.

Числовые типы

Для хранения чисел в Kotlin предусмотрено сразу несколько типов данных. Они отличаются, во-первых, видом чисел, которые могут хранить (целые или с плавающей точкой), а во-вторых, размером (сколько байт памяти занимает каждая переменная).

Целые числа

Для целых чисел есть четыре типа:

Тип	Размер, байт	Минимум	Максимум
Byte	1	-128	+127
Short	2	-32768	+32767
Int	4	-2,147,483,648	+2,147,483,647
Long	8	-9,223,372,036,854,775,808	+9,223,372,036,854,775,807

Чтобы присвоить целое число переменной, можно явно указать тип переменной:

```
val num1: Byte = 1
val num2: Short = 2
val num3: Int = 3
val num4: Long = 4
```

Этот пусть самый надёжный, потому что компилятор точно знает какого типа каждая переменная. Если же тип переменной явно не указан, то компилятор попытается определить тип исходя из присваиваемых значений. Но так как небольшие числа могут подпадать под любой тип, компилятор будет считать их типом `Int`:

```
val num1 = 1 // Это Int
val num2 = 2 // И это Int
val num3 = 3 // Снова Int
val num4 = 4 // Опять Int
```

Исключение сделано для типа `Long` – можно добавить к числу прописную букву `L`, тогда число будет считаться типом `Long`, даже если оно небольшое:

```
val num4 = 4L // А вот тут уже Long!
```

Кроме обычной десятичной формы числа, можно использовать также шестнадцатеричную и двоичную форму, используя префиксы `0x` или `0b`:

```
val numHex = 0x3D
val numBin = 0b00111101
```

Кроме того, для записи больших чисел можно использовать знак подчёркивания (`_`) для визуального разделения чисел на разряды – просто для удобства:

```
val numBig1 = 1_234_567
val numBig2 = 1234567
```

Обе этих переменных будут хранить то же самое число, но визуально первое выглядит более наглядно.

Очень важный момент касается преобразования чисел из одного типа в другой: нельзя просто записать переменную типа `Byte` в переменную типа `Int`! Даже несмотря на то, что любое число из диапазона `Byte` можно представить в диапазоне `Int`. Нужно обязательно явно преобразовать его к новому типу с помощью встроенной функции `toInt`:

```
val num1: Byte = 1
val num2: Int = num1.toInt()
```

То же самое касается и любого другого числового типа. Поэтому для них всех предусмотрены функции `toByte`, `toShort`, `toInt`, `toLong`, `toFloat` и `toDouble` (последние две функции преобразуют целое число в число с плавающей точкой).

Числа с плавающей точкой

Для представления нецелых чисел в компьютерах обычно используется формат с **плавающей точкой**. Это означает, что нецелое число приводится к виду $m \cdot 10^n$. Число m называется **мантиссой**, оно хранит значимые цифры и всегда находится в диапазоне $0 \le m < 10$. Число n называется **порядком**, это степень числа 10, которая отвечает за перемещение десятичной точки в нужную позицию. Любое нецелое число можно привести к такому формату:

- $0.1234 = 1.234 * 10^{-1}$
- $1.234 = 1.234 * 10^{0}$
- $12.34 = 1.234 * 10^{1}$

- $123.4 = 1.234*10^2$

Для хранения чисел с плавающей точкой в Kotlin предусмотрено два типа:

Тип	Размер, байт	Минимум (примерно)	Максимум (примерно)
Float	4	$1.4*10^{-45}$	$3.4*10^{38}$
Double	8	$4.9*10^{-324}$	$1.8*10^{308}$

В языках программирования обычно не используется форма записи числа со знаком умножения и числом 10, вместо этого используется буква `E` (от слова *exponent*):

- 0.1234 = 1.234E-1
- 1.234 = 1.234E0 или 1.234
- 12.34 = 1.234E1
- 123.4 = 1.234E2

Если число присваивается переменной типа `Double`, то можно просто записать число:

```
val num3: Double = 1.2E2
```

Однако для типа `Float` недостаточно указать тип, нужно также обязательно добавлять букву f, чтобы компилятор понимал, что это тип `Float`:

```
val num1: Float = 1.2E2f
```

Кроме того, как и для целых чисел, нужно использовать преобразование типа если переменная одного типа присваивается переменной другого типа:

```
val num1: Float = 1.2E2f
val num2: Double = num1.toDouble()
```

Число с плавающей точкой также можно преобразовать в целое число, при этом дробная часть будет просто отброшена (без всякого округления):

```
val num1: Double = 1.99
val num2: Int = num1.toInt() // Результат: 1
```

Операции с числами

Базовые операции с числами – сложение, вычитание, умножение, деление, а также остаток от деления.

Сложение:

```
val a: Int = 5
val b: Int = 2
val c: Int = a + b
println(c) // 7
```

Вычитание:

```
val c: Int = a - b
println(c) // 3
```

Умножение:

```
val c: Int = a * b
println(c) // 10
```

Деление:

```
val c: Int = a / b
println(c) // 2
```

Если деление выполняется над целыми числами, то результат также будет целым числом; дробная часть, если она была, просто отбрасывается, без округления. Если же операнды были с плавающей точкой, то и результат будет также с дробной частью.

Операция «остаток от деления» (или, как её ещё иногда называют, деление по модулю) позволяет найти остаток от целочисленного деления первого операнда на второй:

```
val a: Int = 37
val b: Int = 5
val c: Int = a % b
println(c) // 2
```

Если остаток от деления равен нулю, то первый операнд кратен второму.

Если операцию нужно проделать над какой-то переменной, то можно использовать сокращенную форму записи, когда знак операции ставится сразу после знака присвоения:

- `a += b // То же самое что a = a + b`
- `a -= b // То же самое что a = a - b`
- `a *= b // То же самое что a = a * b`
- `a /= b // То же самое что a = a / b`
- `a %= b // То же самое что a = a % b`

В этом случае с переменными `a` и `b` будет произведена операция, и результат будет записан в переменную `a`.

В программировании часто используется операция прибавления или вычитания единицы, поэтому для таких операций ввели специальный **унарный оператор**, то есть оператор с одним операндом:

```
var a: Int = 5
a++        // То же самое что a = a + 1
println(a) // 6
a--        // То же самое что a = a - 1
println(a) // Снова 5
```

Математические функции

В стандартные библиотеки Kotlin входит пакет `Math`, содержащий множество полезных математических функций. Большая часть функций принимает в качестве аргумента только числа с плавающей точкой, однако некоторые функции работают и с числами типа `Int`/`Long`:

- `abs` – абсолютное значение числа (включая `Int`/`Long`)
- `sin` / `cos` / `tan` – синус, косинус, тангенс числа
- `asin` / `acos` / `atan` – арксинус, арккосинус, арктангенс числа
- `ceil` / `floor` / `round` / `trunc` – округление числа в большую, меньшую или ближайшую сторону, а также простое отбрасывание дробной части

- `ln` / `log2` / `log10` / `log` – натуральный, двоичный, десятичный логарифм числа, или логарифм по указанному основанию
- `min` / `max` – меньшее или большее из двух чисел (включая `Int`/`Long`)
- `pow` / `sqrt` – возведение числа в степень, квадратный корень числа
- `exp` – экспонента числа

Кроме того, пакет `Math` содержит нужные константы, заданные с достаточной точностью:

- `E` – число *e* (2.71…)
- `PI` – число π (3.14…)

Примеры математических вычислений:

```
val k = Math.sin(Math.PI / 2) * Math.cos(Math.PI * 2)
val m = Math.ceil(Math.exp(k))
```

Символы и строки

За представление текстовой информации в Kotlin отвечают два класса: `Char` и `String`.

Класс `Char` хранит один символ. Это может быть буква, цифра, знак препинания или любой другой символ. В коде программы символы заключаются в одинарные кавычки:

```
val c: Char = 'K'
```

Кроме обычных символов, могут быть и специальные символы, которые напрямую на экране не отображаются, но влияют на отображение других символов. Такие символы записываются с префиксом в виде обратного слеша (`\`). Это, например, символ табуляции \t – он переводит позицию вывода следующего символа в столбец с номером, кратным 8:

```
print('A')
print('\t')
print('B')
print('\t')
print('C')
```

С помощью символа табуляции можно выводить данные в виде таблицы.

Другие служебные символы, которые часто используются вместе – это перевод строки `\n` и возврат каретки `\r`. Они заканчивают текущую строку и переводят позицию вывода в начало следующей строки:

```
print('A')
print('\n')
print('\r')
print('B')
```

Выводить текст по одному символу, конечно, не очень удобно, поэтому для текстов используется отдельный тип `String`, предназначенный для хранения строк. В коде программы строки заключаются в двойные кавычки:

```
val t: String = "Оранжевый апельсин"
```

Строка также может содержать и спецсимволы:

```
val t: String = "Оранжевый\r\nапельсин"
```

```
println(t)
```

В данном примере слово «Оранжевый» будет выведено на одной строке, а слово «апельсин» на другой.

Если нужно соединить несколько строк, то можно использовать операцию сложения, она последовательно соединит строки одну за другой:

```
val t1 = "Это"
val t2 = "строка"
val t3 = t1 + " " + t2
println(t3)
```

Однако в Kotlin переменные можно включать прямо в саму строку, используя знак доллара перед названием переменной:

```
val t3 = "$t1 $t2"
println(t3)
```

В строку можно включать даже переменные других типов, например, числа:

```
val n = 5
val t = "Значение: $n"
println(t)
```

Более того, прямо в строке можно производить манипуляции с данными, заключив выражение в фигурные скобки. Например, можно сложить две переменные, в строку будет вставлен результат этой операции:

```
val a = 2
val b = 7
val sum = "Сумма: ${a + b}"
println(sum)
```

Можно получить отдельный символ строки по его индексу (нумерация начинается с нуля):

```
println(sum[1]) // у
```

Но заменить символ на другой таким способом не получится – строка должна оставаться неизменной.

Строковые и символьные функции

У типов `String` и `Char` есть множество полезных встроенных функций. Ниже приведены самые используемые.

Функция `count` возвращает длину строки в символах:

```
val s = "Буря мглою небо кроет"
println(s.count()) // 21
```

Функция `substring` возвращает фрагмент строки, начиная с указанной позиции:

```
println(s.substring(5)) // мглою небо кроет
```

Можно также указать в функции `substring` второй параметр – до какой позиции нужно брать фрагмент. В этом примере будет взят фрагмент с 5 по 15 позицию:

```
println(s.substring(5, 15)) // мглою небо
```

Есть также родственные функции `substringBefore` (подстрока до указанного символа), `substringAfter` (подстрока после указанного символа), `substringBeforeLast` (подстрока до указанного последнего символа) и `substringAfterLast` (подстрока после последнего указанного символа). В следующем примере будет показан фрагмент текста после первого встретившегося пробела:

```
println(s.substringAfter(' ')) // мглою небо кроет
```

Ещё две похоже функции – `take`, которая возвращает первые символы, и `takeLast`, которая возвращает последние символы строки:

```
println(s.take(10))     // Буря мглою
println(s.takeLast(10)) // небо кроет
```

Функция `indexOf` возвращает позицию в строке, где находится указанный символ (или строка). Например:

```
println(s.indexOf('н'))     // 11
println(s.indexOf("мглою")) // 5
```

Функция reversed возвращает строку, записанную в обратном порядке – иногда это тоже может быть полезно:

```
println(s.reversed()) // теорк обен юолгм яруБ
```

Функции `padStart` и `padEnd` дополняют начало или конец строки указанными символами до указанной длины (так, чтобы вместе с добавленными символами новая строка была не короче указанного количества символов):

```
val t = "1"
println(t.padStart(5, '0')) // 00001
println(t.padEnd(3, '0'))   // 100
```

Функции `drop` и `dropLast`, напротив, удаляют указанное количество символов из начала или конца строки:

```
val t = "1234567890"
println(t.drop(3))     // 4567890
println(t.dropLast(3)) // 1234567
```

Можно также удалить пробелы в начале и конце строки – иногда это бывает полезно:

```
val t = "  зима  "
println(t.trim()) // зима
```

Функция `replace` заменяет все вхождения указанной подстроки на другую подстроку:

```
val t = "яблоко, ещё яблоко, и груша"
println(t.replace("яблоко", "апельсин")) // апельсин, ещё апельсин, и груша
```

Строку или символ можно перевести в верхний или нижний регистр:

```
val t = "абвгд"
println(t.toUpperCase()) // АБВГД
val c = 'Z'
```

```
println(c.toLowerCase()) // z
```

А для символа можно также использовать функции `isUpperCase` и `isLowerCase`, которые проверяют является ли символ прописным или строчным:

```
val c = 'z'
println(c.isUpperCase()) // false
println(c.isLowerCase()) // true
```

Наконец, если в строке записано число, то можно перевести его в один из числовых типов:

```
val str1 = "123"
val n = str1.toInt()
val str2 = "1.23"
val d = str2.toDouble()
```

А число можно таким же образом перевести в строку:

```
val str3 = d.toString()
```

Логический тип

Для представления логических значений используется тип `Boolean`. Переменная такого типа может содержать лишь два значения: `true` (истина) или `false` (ложь):

```
val b: Boolean = true
```

Чаще всего логический тип появляется в результате сравнения значений. Например:

```
val n = 5
val m = 7
val b1 = n == m // false
val b2 = n > m  // false
val b3 = n < m  // true
```

Для операций с логическим типом есть специальные логические операторы. Оператор *логическое НЕ*, который представлен префиксом в виде восклицательного знака, меняет значение на противоположное: `true` на `false`, а `false` на `true`.

Операнд	Результат
true	**false**
false	**true**

Пример:

```
val b1 = !(n == m) // true
```

Оператор *логическое И* (`and`) возвращает `true` только в том случае, если оба операнда равны `true`:

Операнд 1	Операнд 2	Результат
true	true	**true**
true	false	**false**
false	true	**false**

false	false	**false**

Пример:

```
val b1 = (n == m) and (n > 3) // false
val b2 = (m < 10) and (n > 3) // true
```

Оператор *логическое ИЛИ* (`or`) возвращает `true` если хотя бы один из операндов равен `true`:

Операнд 1	Операнд 2	Результат
true	true	**true**
true	false	**true**
false	true	**true**
false	false	**false**

Пример:

```
val b1 = (n == m) and (n > 3) // true
val b2 = (m < 10) and (n > 3) // true
val b3 = (m < 3) and (n > 10) // false
```

Оператор *логическое исключающее ИЛИ* (`xor`) возвращает true если один из операторов равен `true`, а другой `false`:

Операнд 1	Операнд 2	Результат
true	true	**false**
true	false	**true**
false	true	**true**
false	false	**false**

Пример:

```
val b1 = (n == m) and (n > 3) // true
val b2 = (m < 10) and (n > 3) // false
```

Массивы

Массив – это структура, в которой хранятся данные одного типа. Массив похож на строку, но если в строке в каждой ячейке хранится символ типа `Char`, то в массиве в каждой ячейке может храниться значение любого типа (но одного и того же во всём массиве).

Для объявления массива используется класс `Array`. Это **обобщённый класс** – то есть класс, который умеет работать с разными типами данных – числами, строками, объектами и т. д. Однако тип данных массива задаётся уже на этапе компиляции программы, поменять его не получится, поэтому тип данных должен быть либо указан явно, либо определен по элементам массива. Явное указание типа:

```
val arr: Array<Int>
```

Здесь объявляется переменна `arr`, с помощью которой можно будет получать доступ к элементам массива. Хранить в таком массиве можно будет целые числа типа `Int`.

Хотя переменная объявлена, но массив пока пустой, в нём нет ни одного значения. Инициализировать массив можно несколькими способами. Самый простой – использовать функцию `arrayOf`:

```
arr = arrayOf(1, 2, 3)
```

Здесь создаётся массив из трёх целых чисел [1, 2, 3], и ссылка на него записывается в переменную `arr`.

Кроме прямого перечисления элементов, можно инициализировать массив с помощью генерации элементов по индексу:

```
arr = Array<Int>(5) { i -> i * 2 }
```

В этом примере создаётся массив из пяти целых чисел, а затем выполняется код в фигурных скобках. Слева от стрелки стоит переменная, в которую будут последовательно записываться индексы каждого элемента массива (0, 1, … 4), а справа от стрелки стоит выражение, которое будет вычислено и записано в очередной элемент массива. В примере индекс элемента просто умножается на 2, то есть значениями массива станут [0, 2, 4, 6, 8].

Наконец, можно заполнить весь массив одним и тем же элементом:

```
arr = Array<Int>(7, {10})
```

В данном примере будет создан массив из 7 элементов типа `Int`, и в каждый элемент будет записано число 10.

Получить доступ к любому элементу массива можно с помощью квадратных скобок с индексом элемента. Причем, в отличие от строк, элемент массива можно изменить:

```
println(arr[1])
arr[0] = 10
```

Помимо универсального класса `Array` есть также специализированные классы для массивов определенного типа. Например, класс `IntArray` позволяет работать только с массивами с элементами типа `Int`. Для наполнения такого массива нужно использовать соответствующие функции, например, `intArrayOf`:

```
var ia: IntArray = intArrayOf(1, 2, 3)
```

Хотя такой массив целых чисел тоже является массивом, однако он не совместим с обычным массивом типа `Array`, и нельзя смешивать функции массивов общего типа и специализированных массивов. Поэтому во избежание путаницы лучше использовать обобщенный класс `Array` и функцию `arrayOf`.

Функции для работы с массивами

Kotlin содержит много полезных функций для работы с массивами. Некоторые из них аналогичны по смыслу функциям для работы со строками, описанными выше.

- `count` – количество элементов в массиве
- `average`, `sum` – подсчёт среднего значения или суммы элементов для числовых массивов
- `distinct` – возвращает список неповторяющихся элементов массива

- `drop`, `dropLast` – возвращает список элементов массива без указанного количества первых или последних элементов
- `take`, `takeLast` – возвращает указанное количество первых или последних элементов массива
- `indexOf` – возвращает позицию в массиве искомого элемента (или -1 если элемент не найден)
- `reversed` – возвращает список элементов массива в обратном порядке
- `shuffle` – возвращает список элементов массива в случайном порядке
- `sorted` – возвращает отсортированный список элементов массива

Массивы не всегда являются наилучшим выбором для представления больших последовательностей однотипных данных, кроме них есть ещё списки, наборы, словари и другие варианты хранения данных. Они будут рассмотрены позже.

Последовательности

Диапазон значений в Kotlin называется последовательностью. Последовательность – это такой же тип данных, как и остальные, только содержит сразу несколько значений. Задаётся последовательность первым и последним значениями, разделёнными двумя точками:

```
val q = 1..5
```

Теперь переменная q будет содержать целые числа от 1 до 5, которые можно, например, перебрать с помощью цикла.

Начало последовательности при таком формате всегда должно быть меньше её окончания, если задать последовательность как `5..1`, то она будет пустой, элементов в ней не будет. Чтобы задать последовательность в сторону уменьшения, нужно использовать форму записи с оператором `downTo`:

```
val q = 5 downTo 1
```

Можно задать шаг, с которым будет увеличиваться начальное значение:

```
val q = 1..10 step 2
```

В этом примере последовательность будет содержать числа 1, 3, 5, 7 и 9.

Можно задать последовательность не включая правую границу:

```
val q = 1 until 10
```

Теперь в последовательности будут числа от 1 до 9.

Последовательность может содержать не только числа, но и символы:

```
val q = 'a'..'f'
```

Теперь последовательность будет содержать символы a, b, c, d, e и f.

Можно задать последовательность из произвольных элементов:

```
val q = sequenceOf(1, 10, 100, 1000)
```

Наконец, можно преобразовать массив в последовательность с помощью метода `asSequence`:

```
val a = arrayOf(1, 7, 3, 9, 8)
val q = a.asSequence()
```

Теперь последовательность будет содержать те же числа, что и массив: 1, 7, 3, 9, 8, причем именно в этом же порядке.

Чтобы проверить входит ли значение в последовательность, используется оператор `in`:

```
println(100 in q) // true
```

Напротив, оператор `!in` проверяет что значение не входит в последовательность:

```
println(100 !in q) // false
```

Управление потоком

Оператор if

Оператор `if` позволяет выполнить блок программы в зависимости от того истинно ли некоторое условие:

```
if (k > 7) {
    // Действия если условие истинно
    println("Да, k больше 7!")
}
```

Условие задаётся в круглых скобках после оператора `if`, и может включать различные сравнения, или другие выражения, которые возвращают логическое значение `true` или `false`. Например:

```
if (k == 5) { ... }
if (k > 5) { ... }
if ((k >= 1) and (k <= 10)) { ... }
if (k in 20..30) { ... }
```

Для проверки равенства в Kotlin, как и в других языках программирования, используется двойной знак равенства (`==`).

Иногда требуется выполнить одно действие не только если условие истинно, но и какое-то другое действие если условие ложно, в этом случае можно использовать дополнительный оператор else:

```
if (k > 7) {
    // Действия если условие истинно
    println("Да, k больше 7!")
}
else {
    // Действия если условие ложно
    println("Нет, k не больше 7...")
}
```

Если действие состоит только из одного оператора, то фигурные скобки можно опустить, хоть это иногда и может снижать визуальную понятность кода:

```
if (k > 7)
```

```
    println("Да, k больше 7!")
else
    println("Нет, k не больше 7...")
```

Оператор `if` в Kotlin может и возвращать значения в зависимости от условий:

```
val color = if (answer == 1) "Красный" else "Зеленый"
```

Этот компактный код эквивалентен следующему коду, который выполняет те же самые действия:

```
val color: String
if (answer == 1)
    color = "Красный"
else
    color = "Зеленый"
```

В других языках программирования для таких же целей используется тернарный оператор, но в Kotlin в нём нет необходимости, так как оператор `if` может выполнять те же функции.

Оператор when

Оператор `when` позволяет выполнить действия в зависимости от выполнения условий. Он похож на оператор `if`, но включает сразу несколько выражений:

```
when (answer) {
    1 -> println("Красный")
    2 -> println("Зеленый")
    3 -> println("Синий")
    else -> println("Серобуромалиновый")
}
```

Этот код можно записать и с помощью оператора `if`, но в более громоздкой и менее наглядной форме:

```
if (answer == 1)
    println("Красный")
else if (answer == 2)
    println("Зеленый")
else if (answer == 3)
    println("Синий")
else
    println("Серобуромалиновый")
```

Блок `else`, который выполняется если ни одно условие не оказалось истинным, не обязателен, можно его и не включать:

```
when (answer) {
    1 -> println("Красный")
    2 -> println("Зеленый")
    3 -> println("Синий")
}
```

Условия могут включать проверку на вхождение в последовательность (с помощью оператора `in`), а также перечисление сразу нескольких значений через запятую:

```
when (answer) {
    1, 2, 3 -> println("Правильный цвет")
    in 4..10 -> println("Подозрительный цвет")
    else -> println("Не знаю такого цвета!")
}
```

Оператор `when`, как и оператор `if`, можно использовать для присвоения значений:

```
val color = when (answer) {
    1, 2, 3 -> "Правильный цвет"
    in 4..10 -> "Подозрительный цвет"
    else -> "Не знаю такого цвета!"
}
```

В этом случае блок `else` является обязательным, так как какое-то значение в переменную нужно записать в любом случае.

Оператор for

Оператор `for` предназначен для перебора значений: это могут быть элементы массива, последовательности, списка и т. д.:

```
for (n in 1..5) {
    print("$n ")
}
```

Оператор `print` в этом коде выполнится 5 раз, и на каждом шаге переменная `n` будет принимать очередное значение последовательность `1..5`: при первом выполнении `n` будет равна 1, при втором – 2 и т. д. В итоге будет напечатан ряд чисел 1 2 3 4 5.

Таким образом, цикл `for` скорее похож на цикл `foreach` из других языков программирования, чем обычный цикл `for`.

Особенности цикла `for` для перебора массивов разобраны ниже.

Оператор while

Оператор `while` выполняет блок кода до тех пор, пока условное выражение истинно:

```
var i = 1
while (i < 10) {
    print("$i ")
    i++
}
```

В этом примере используется дополнительная переменная `i`, которой перед циклом присваивается значение 1. После оператора `while` в круглых скобках записано условие, при котором цикл будет повторяться: пока `i` меньше 10.

В отличие от цикла `for`, в цикле while переменная `i` автоматически не изменяется, поэтому нужно не забывать самостоятельно изменять её в теле цикла, например, увеличивая на единицу (`i++`). В противном случае цикл никогда не закончится, и программа зависнет.

Существует также другая форма записи цикла, в которой условие стоит не в начале цикла, а в конце – такой цикл называется **циклом с постусловием**:

```
var i = 1
do {
    print("$i ")
    i++
} while (i < 10)
```

В цикле с постусловием тело цикла всегда выполнится хотя бы один раз, и только потом будет проверена истинность выражения. А в **цикле с предусловием**, где выражение стоит в начале, тело цикла иногда не выполнится ни разу – в том случае, если выражение изначально имеет ложное значение.

Прерывание циклов

Иногда нужно досрочно завершить работу цикла – например, если искомое значение найдено и в дальнейшем просмотре массива нет смысла. Оператор `break` прерывает работу текущего цикла и передаёт управление за его пределы:

```
for (n in 1..10) {
    println(n)
    if (n == 5)
        break
}
```

Этот цикл будет выводить числа от 1 до 10, но когда встретится число 5 – цикл будет прерван, поэтому в консоли появятся только цифры от 1 до 5.

Если один цикл вложен в другой, то оператор `break` прервёт только тот цикл, в теле которого сам оператор находится, а внешний цикл будет продолжать выполняться. Например, следующий код распечатает таблицу умножения, но после 8 столбца остаток строки каждый раз не будет выводиться, однако следующие строки будут продолжать печататься:

```
for (n in 1..9) {
    for (m in 1..9) {
        print("${n * m}\t")
        if (m == 8)
            break
    }
    println()
}
```

Другой оператор `continue` позволяет прервать текущую итерацию цикла и перейти к следующей:

```
for (n in 1..9) {
    if (n == 5)
        continue
    print("$n ")
}
```

В этом примере в консоль будут выводиться числа от 1 до 9, однако если встретится число 5, то этот шаг цикла будет завершен, и цикл перейдёт к следующему значению. Поэтому на экране появятся числа 1 2 3 4 6 7 8 9.

Перебор значений

С помощью циклов for и while можно перебирать значения массивов, последовательностей, списков и т. д. С помощью цикла for можно перебирать сами значения, например, в массиве:

```
val fruits = arrayOf("апельсин", "груша", "яблоко", "вишня", "арбуз")
for (fruit in fruits) {
    println(fruit)
}
```

На каждом шаге переменная цикла `fruit` будет получать очередное значение массива. Однако в ряде случаев бывает полезно не просто получить значение, но и знать его позицию в массиве. В этом случае удобнее использовать цикл `while`, в котором переменная используется как индекс:

```
var i = 0
while (i < fruits.count()) {
    println(fruits[i])
    i++
}
```

Здесь уже нет переменной `fruit`, в которую на очередном шаге записывается значение: для доступа к значению используется имя массива и индекс в квадратных скобках `fruits[i]`.

Цикл `while` в этом примере выполняется до тех пор, пока переменная `i` не дойдёт до последнего элемента. Количество элементов в массиве возвращает функция `count`, но так как элементы массива нумеруются с 0, то и переменная `i` должна доходить только до значения на единицу меньше.

Массивы, как и другие последовательности, имеют также встроенную функцию `forEach`, которая позволяет перебирать элементы по одному:

```
fruits.forEach({ it -> println(it) })
```

Подробнее такая форма записи будет разобрана позже, в разделе лямбда-функций.