Основы языка Kotlin

Структура программы

Функция main Точкой входа в программу на языке Kotlin является функция main. Именно с этой функции начинается выполнение программы на Kotlin, поэтому эта функция должна быть в любой программе на языке Kotlin.

Так, в прошлой теме была определена следующая функция main:

```
fun main(){
    println("Hello Kotlin")
}
```

Определение функции main() (в принципе как и других функций в Kotlin) начинается с ключевого слова fun. По сути оно указывает, что дальше идет определение функции. После fun указывается имя функции. В данном случае это main.

После имени функции в скобках идет список параметров функции. Здесь функция main не принимает никаких параметров, поэтому после имени функции идут пустые скобки.

Все действия, которые выполняет функция, заключаются в фигурные скобки. В данном случае единственное, что делает функция main, - вывод на консоль некоторого сообщения с помощью другой встроенной функции println().

Стоит отметить, что до версии 1.3 в Kotlin функция main должна была принимать параметры:

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

Параметр args: Array представляет массив строк, через который в программу можно передать различные данные.

Начиная с версии 1.3 использовать это определение функции с параметрами необязательно. Хотя мы можем его использовать.

Инструкции и блоки кода

Основным строительным блоком программы на языке Kotlin являются инструкции (statement). Каждая инструкция выполняет некоторое действие, например, вызовы функций, объявление переменных и присвоение им значений. Например:

```
println("Hello Kotlin!");
```

Данная строка представляет встроенной функции println(), которая выводит на консоль, некоторое сообщение (в данном случае строку "Hello Kotlin!").

Стоит отметить, что в отличие от других похожих языков программирования, например, Java, в Kotlin не обязательно ставить после инструкции точку запятой. Каждая инструкция просто размещается на новой строке:

```
fun main(){
    println("Kotlin on Metanit.com")
    println("Hello Kotlin")
    println("Kotlin is a fun")
}
```

Тем не менее, если инструкции располагаются на одной строке, то чтобы их отделить друг от друга, надо указывать после инструкции точку с запятой:

```
fun main(){
    println("Kotlin on Metanit.com");println("Hello Kotlin");println("Kotlin is a
fun")
}
```

Комментарии

Код программы может содержать комментарии. Комментарии позволяют понять смысл программы, что делают те или иные ее части. При компиляции комментарии игнорируются и не оказывают никакого влияния на работу приложения и на его размер.

В Kotlin есть два типа комментариев: однострочный и многострочный. Однострочный комментарий размещается на одной строке после двойного слеша //. А многострочный комментарий заключается между символами /* текст комментария */. Он может размещаться на нескольких строках. Например:

```
/*
    многострочный комментарий
    Функция main -
    точка входа в программу
*/
fun main(){         // начало функции main

    println("Hello Kotlin") // вывод строки на консоль
}                   // конец функции main
```

Переменные

Для хранения данных в программе в Kotlin, как и в других языках программирования, применяются переменные. Переменная представляет именованный участок памяти, который хранит некоторое значение.

Каждая переменная характеризуется определенным именем, типом данных и значением. Имя переменной представляет поизвольный идентификатор, который может содержать алфавитно-цифровые символы или символ подчеркивания и должен начинаться либо с алфавитного символа, либо со знака подчеркивания. Для определения переменной можно использовать либо ключевое слово val, либо ключевое слово var.

Формальное определение переменной:

```
val|var имя_переменной: тип_переменной
```

Вначале идет слово val или var, затем имя переменной и через двоеточие тип переменной.

Например, определим переменную age:

```
val age: Int
```

То есть в данном случае объявлена переменная age, которая имеет тип Int. Тип Int говорит о том, что переменная будет содержать целочисленные значения.

После определения переменной ей можно присвоить значение:

```
fun main() {
    val age: Int
    age = 23
    println(age)
}
```

Для присвоения значения переменной используется знак равно. Затем мы можем производить с переменной различные операции. Например, в данном случае с помощью функции println() значение переменной выводится на консоль. И при запуске этой программы на консоль будет выведено число 23.

Присвоение значения переменной должно производиться только после ее объявления. И также мы можем сразу присвоить переменной начальное значение при ее объявлении. Такой прием называется инициализацией:

```
fun main() {
    val age: Int = 23
    println(age)
}
```

Однако обязательно надо присвоить переменной некоторое значение до ее использования:

```
fun main() {

    val age: Int
    println(age)// Ошибка, переменная не инициализирована
}
```

Изменяемые и неизменяемые переменные

Выше было сказано, что переменные могут объявляться как с помощью слова val, так и с помощью слова var. В чем же разница между двумя этими способами?

С помощью ключевого слова val определяется неизменяемая переменная (immutable variable). То есть мы можем присвоить значение такой переменной только один раз, но изменить его после первого присвоения мы уже не сможем. Например, в следующем случае мы получим ошибку:

```
fun main() {
    val age: Int
    age = 23        // здесь норм - первое присвоение
    age = 56        // здесь ошибка - переопределить значение переменной нельзя
    println(age)
}
```

А у переменной, которая определена с помощью ключевого слова var мы можем многократно менять значения (mutable variable):

```
fun main() {
    var age: Int
    age = 23
    println(age)
    age = 56
    println(age)
}
```

Поэтому если не планируется изменять значение переменной в программе, то лучше определять ее с ключевым словом val.

Определение констант

Также Kotlin поддерживает константы времени компиляции. Для их определения применяются ключевые слова const val:

```
const val maxAge = 120  // константа
fun main() {
    println(maxAge)
}
```

В данном случае maxAge является константой.

Отличительной особенностью констант является то, что они на стадии компиляции должны иметь некоторое значение, и это значение изменить нельзя. Это накладывает на использование констант ряд ограничений:

- Естественно нельзя изменить значение константы:

```
const val maxAge = 120  // константа
fun main() {
    maxAge = 1500   // ошибка
    println(maxAge)
}
```

Здесь при попытке присвоения константе maxAge нового значения в функции main мы столкнемся с ошибкой на стадии компиляции. Если мы работаем в среде Intellij IDEA или Android Studio, то уже при написании кода на подобные ошибки укажет сама среда разработки.

- Константа должна объявляться на самом верхнем уровне (вне класса/функции):

```
fun main() {
    const val maxAge = 120   // ошибка
    println(maxAge)
}
```

- Тип данных константы должен соответствовать одному из примитивных (например, Int) или типу String

Также стоит отметит отличие val-переменных от констант (const val): значение val-переменных устанавливается во время выполнения, а значение констант - во время компиляции. Значение val-переменной также нельзя изменить после установки, однако мы можем объявить val-переменную, а потом дальше в программе присвоить ей значение. Константе же необходимо присвоить значение сразу при определении.

Типы данных

В Kotlin все компоненты программы, в том числе переменные, представляют объекты, которые имеют определенный тип данных. Тип данных определяет, какой размер памяти может занимать объект данного типа и какие операции с ним можно производить. В Kotlin есть несколько базовых типов данных: числа, символы, строки, логический тип и массивы.

Целочисленные типы

- Byte: хранит целое число от -128 до 127 и занимает 1 байт
- Short: хранит целое число от -32 768 до 32 767 и занимает 2 байта
- Int: хранит целое число от -2 147 483 648 (-231) до 2 147 483 647 (231 - 1) и занимает 4 байта
- Long: хранит целое число от –9 223 372 036 854 775 808 (-263) до 9 223 372 036 854 775 807 (263-1) и занимает 8 байт

В последней версии Kotlin также добавлена поддержка для целочисленных типов без знака:

- UByte: хранит целое число от 0 до 255 и занимает 1 байт
- UShort: хранит целое число от 0 до 65 535 и занимает 2 байта
- UInt: хранит целое число от 0 до 232 - 1 и занимает 4 байта
- ULong: хранит целое число от 0 до 264-1 и занимает 8 байт

Объекты целочисленных типов хранят целые числа:

```
fun main(){

    val a: Byte = -10
    val b: Short = 45
    val c: Int = -250
    val d: Long = 30000
    println(a) // -10
    println(b) // 45
    println(c) // -250
    println(d) // 30000
}
```

Для передачи значений объектам, которые представляют беззнаковые целочисленные типы данных, после числа указывается суффикс U:

```
fun main(){

    val a: UByte = 10U
    val b: UShort = 45U
    val c: UInt = 250U
    val d: ULong = 30000U
    println(a) // 10
    println(b) // 45
    println(c) // 250
    println(d) // 30000
}
```

Кроме чисел в десятичной системе мы можем определять числа в двоичной и шестнадцатеричной системах.

Шестнадцатеричная запись числа начинается с 0x, затем идет набор символов от 0 до F, которые представляют число:

```
val address: Int = 0x0A1    // 161
println(address) // 161
```

Двоичная запись числа предваряется символами 0b, после которых идет последовательность из нулей и единиц:

```
val a: Int = 0b0101    // 5
val b: Int = 0b1011     // 11
println(a)      // 5
println(b)      // 11
```

Числа с плавающей точкой Кроме целочисленных типов в Kotlin есть два типа для чисел с плавающей точкой, которые позволяют хранить дробные числа:

- Float: хранит число с плавающей точкой от -3.4*1038 до 3.4*1038 и занимает 4 байта
- Double: хранит число с плавающей точкой от ±5.0*10-324 до ±1.7*10308 и занимает 8 байта.

В качестве разделителя целой и дробной части применяется точка:

```
val height: Double = 1.78
val pi: Float = 3.14F
println(height)      // 1.78
println(pi)         // 3.14
```

Чтобы присвоить число объекту типа Float после числа указывается суффикс f или F.

Также тип Double поддерживает экспоненциальную запись:

```
val d: Double = 23e3
println(d)      // 23 000

val g: Double = 23e-3
println(g)      // 0.023
```

Логический тип Boolean

Тип Boolean может хранить одно из двух значений: true (истина) или false (ложь).

```
val a: Boolean = true
val b: Boolean = false
```

Символы

Символьные данные представлены типом Char. Он представляет отдельный символ, который заключается в одинарные кавычки.

```
val a: Char = 'A'
val b: Char = 'B'
val c: Char = 'T'
```

Также тип Char может представлять специальные последовательности, которые интерпретируются особым образом:

- \t: табуляция
- \n: перевод строки
- \r: возврат каретки
- ': одинарная кавычка
- ": двойная кавычка
- \: обратный слеш

Строки

Строки представлены типом String. Строка представляет последовательность символов, заключенную в двойные кавычки, либо в тройные двойные кавычки.

```
fun main() {

    val name: String = "Eugene"

    println(name)
}
```

Строка может содержать специальные символы или эскейп-последовательности. Например, если необходимо вставить в текст перевод на другую строку, можно использовать эскейп-последовательность \n:

```
val text: String = "SALT II was a series of talks between United States \n and
Soviet negotiators from 1972 to 1979"
```

Для большего удобства при создании многострочного текста можно использовать тройные двойные кавычки:

```
fun main() {

    val text: String = """
                        SALT II was a series of talks between United States
                        and Soviet negotiators from 1972 to 1979.
                        It was a continuation of the SALT I talks.
                    """
    println(text)
}
```

Шаблоны строк

Шаблоны строк (string templates) представляют удобный способ вставки в строку различных значений, в частности, значений переменных. Так, с помощью знака доллара $ мы можем вводить в строку значения различных переменных:

```
fun main() {

    val firstName = "Tom"
    val lastName = "Smith"
    val welcome = "Hello, $firstName $lastName"
    println(welcome)    // Hello, Tom Smith
}
```

В данном случае вместо `$firstName` и `$lastName` будут вставляться значения этих переменных. При этом переменные необязательно должны представлять строковый тип:

```
val name = "Tom"
val age = 22
val userInfo = "Your name: $name  Your age: $age"
```

Выведение типа

Kotlin позволяет выводить тип переменной на основании данных, которыми переменная инициализируется. Поэтому при инициализации переменной тип можно опустить:

```
val age = 5
```

В данном случае компилятор увидит, что переменной присваивается значение типа Int, поэтому переменная age будет представлять тип Int.

Соответственно если мы присваиваем переменной строку, то такая переменная будет иметь тип String.

```
val name = "Tom"
```

Любые целые числа, воспринимаются как данные типа Int.

Если же мы хотим явно указать, что число представляет значение типа Long, то следует использовать суффикс L:

```
val sum = 45L
```

Если надо указать, что объект представляет беззнаковый тип, то применяется суффикс u или U:

```
val sum = 45U
```

Аналогично все числа с плавающей точкой (которые содержат точку в качестве разделителя целой и дробной части) рассматриваются как числа типа Double:

```
val height = 1.78
```

Если мы хотим указать, что данные будут представлять тип Float, то необходимо использовать суффикс f или F:

```
val height = 1.78F
```

Однако нельзя сначала объявить переменную бз указания типа, а потом где-то в программе присвоить ей какое-то значение:

```
val age     // Ошибка, переменная не инициализирована
age = 5
```

Статическая типизация и тип Any

Тип данных ограничивает набор значений, которые мы можем присвоить переменной. Например, мы не можем присвоить переменной типа Double строку:

```
val height: Double = "1.78"
```

И после того, как тип переменной установлен, он не может быть изменен:

```
fun main() {

    var height: String = "1.78"
    height = 1.81       // !Ошибка - переменная height хранит только строки
    println(height)
}
```

Однако в Kotlin также есть тип Any, который позволяет присвоить переменной данного типа любое значение:

```
fun main() {

    var name: Any = "Tom"
    println(name)   // Tom
    name = 6758
    println(name)   // 6758
}
```

Консольный ввод и вывод

Вывод на консоль

Для вывода информации на консоль в Kotlin есть две встроенные функции:

```
print()
println()
```

Обе эти функции принимают некоторый объект, который надо вывести на консоль, обычно это строка. Различие между ними состоит в том, что функция println() при выводе на консоль добавляет перевод на новую строку:

```
fun main() {

    print("Hello ")
    print("Kotlin ")
```

```
    print("on Metanit.com")
    println()
    println("Kotlin is a fun")
}
```

Причем функция println() необязательно должна принимать некоторое значения. Так, здесь применяется пустой вызов функции, который просто перевод консольный вывод на новую строку:

```
println()
```

Консольный вывод программы:

```
Hello Kotlin on Metanit.com
Kotlin is a fun
```

Ввод с консоли

Для ввода с консоли применяется встроенная функция readLine(). Она возвращает введенную строку. Стоит отметить, что результат этой функции всегда представляет объект типа String. Соответственно введеную строку мы можем передать в переменную типа String:

```
fun main() {

    print("Введите имя: ")
    val name = readLine()

    println("Ваше имя: $name")
}
```

Здесь сначала выводится приглашение к вводу данных. Далее введенное значение передается в переменную name. Результат работы программы:

```
Введите имя: Евгений
Ваше имя: Евгений
```

Подобным образом можно вводить разные данные:

```
fun main() {

    print("Введите имя: ")
    val name = readLine()
    print("Введите email: ")
```

```
    val email = readLine()
    print("Введите адрес: ")
    val address = readLine()

    println("Ваше имя: $name")
    println("Ваш email: $email")
    println("Ваш адрес: $address")
}
```

Пример работы программы:

```
Введите имя: Евгений
Введите email: metanit22@mail.ru
Введите адрес: ул. Кленов, д.31, кв. 20
Ваше имя: Евгений
Ваш email: metanit22@mail.ru
Ваш адрес: ул. Кленов, д.31, кв. 20
```

Операции с числами

Арифметические операции

Kotlin поддерживает базовые арифметические операции:

- (сложение): возвращает сумму двух чисел.

```
val x = 5
val y = 6
val z = x + y
println(z)      // z = 11
```

- - (вычитание): возвращает разность двух чисел.

```
val x = 5
val y = 6
val z = x - y  // z = -1
```

- * (умножение): возвращает произведение двух чисел.

```
val x = 5
val y = 6
val z = x * y  // z = 30
```

- / (деление): возвращает частное двух чисел.

```
val x = 60
val y = 10
val z = x / y  // z = 6
```

При этом если в операции деления оба операнда представляют целые числа, то результатом тоже будет целое число, а если в процессе деления образовалась дробная часть, то она отбрасывается:

```
fun main() {

    val x = 11
    val y = 5
    val z = x / y  // z =2
    println(z)// 2
}
```

Так в данном случае, хотя если согласно стандартной математике разделить 11 на 5, то получится 2.2. Однако поскольку оба операнда представляют целочисленный тип, а именно тип Int, то дробная часть - 0.2 отрабрасывается, поэтому результатом будет число 2, а переменная z будет представлять тип Int.

Чтобы результатом было дробное число, один из операндов должен представлять число с плавающей точкой:

```
fun main() {

    val x = 11
    val y = 5.0
    val z = x / y   // z =2.2
    println(z)      // 2.2
}
```

В данном случае переменная y представляет тип Double, поэтому результатом деления будет число 2.2, а переменная z также будет представлять тип Double.

- %: возвращает остаток от целочисленного деления двух чисел.

```
val x = 65
val y = 10
val z = x % y  // z = 5
```

- ++ (инкремент): увеличивает значение на единицу.

Префиксный инкремент возвращает увеличенное значение:

```
var x = 5
val y = ++x
println(x)      // x = 6
println(y)      // y = 6
```

Постфиксный инкремент возвращает значение до увеличения на единицу:

```
var x = 5
val y = x++
println(x)      // x = 6
println(y)      // y = 5
```

- -- (декремент): уменьшает значение на единицу.

Префиксный декремент возвращает уменьшенное значение:

```
var x = 5
val y = --x
println(x)      // x = 4
println(y)      // y = 4
```

Постфиксный декремент возвращает значение до уменьшения на единицу:

```
var x = 5
val y = x--
println(x)      // x = 4
println(y)      // y = 5
```

Также есть ряд операций присвоения, которые сочетают арифметические операции и присвоение:

- +=: присваивание после сложения. Присваивает левому операнду сумму левого и правого операндов: A += B эквивалентно A = A + B
- -=: присваивание после вычитания. Присваивает левому операнду разность левого и правого операндов: A -= B эквивалентно A = A - B
- *=: присваивание после умножения. Присваивает левому операнду произведение левого и правого операндов: A *= B эквивалентно A = A * B
- /=: присваивание после деления. Присваивает левому операнду частное левого и правого операндов: A /= B эквивалентно A = A / B
- %=: присваивание после деления по модулю. Присваивает левому операнду остаток от целочисленного деления левого операнда на правый: A %= B эквивалентно A = A % B

Поразрядные операции

Ряд операций выполняется над двоичными разрядами числа. Здесь важно понимать, как выглядит двоичное представление тех или иных чисел. В частности, число 4 в двоичном виде - 100, а число 15 - 1111.

Есть следующие поразрядные операторы (они применяются только к данным типов Int и Long):

- shl: сдвиг битов числа со знаком влево

```
val z = 3 shl 2     // z = 11 << 2 = 1100
println(z)          // z = 12
val d = 0b11 shl 2
println(d)          // d = 12
```

В данном случае число сдвигается на два разряда влево, поэтому справа число в двоичном виде дополняется двумя нулями. То есть в двоичном виде 3 представляет 11. Сдвигаем на два разряда влево (дополняем справа двумя нулями) и получаем 1100, то есть в десятичной системе число 12.

- shr: сдвиг битов числа со знаком вправо

```
val z = 12 shr 2     // z = 1100 >> 2 = 11
println(z)          // z = 3
val d = 0b1100 shr 2
println(d)          // d = 3
```

Число 12 сдвигается на два разряда вправо, то есть два числа справа факически отбрасываем и получаем число 11, то есть 3 в десятичой системе.

- ushr: сдвиг битов беззнакового числа вправо

```
val z = 12 ushr 2     // z = 1100 >> 2 = 11
println(z)          // z = 3
```

- and: побитовая операция AND (логическое умножение или конъюнкция). Эта операция сравнивает соответствующие разряды двух чисел и возвращает единицу, если эти разряды обоих чисел равны 1. Иначе возвращает 0.

```
val x = 5   // 101
val y = 6   // 110
val z = x and y     // z = 101 & 110 = 100
println(z)          // z = 4

val d = 0b101 and 0b110
println(d)          // d = 4
```

- or: побитовая операция OR (логическое сложение или дизъюнкция). Эта операция сравнивают два соответствуюших разряда обоих чисел и возвращает 1, если хотя бы один разряд равен 1. Если оба разряда равны 0, то возвращается 0.

```
val x = 5   // 101
val y = 6   // 110
val z = x or y     // z = 101 | 110 = 111
println(z)         // z = 7

val d = 0b101 or 0b110
println(d)          // d = 7
```

- xor: побитовая операция XOR. Сравнивает два разряда и возвращает 1, если один из разрядов равен 1, а другой равен 0. Если оба разряда равны, то возвращается 0.

```
val x = 5   // 101
val y = 6   // 110
val z = x xor y     // z = 101 ^ 110 = 011
println(z)         // z = 3

val d = 0b101 xor 0b110
println(d)          // d = 3
```

- inv: логическое отрицание или инверсия - инвертирует биты числа

```
val b = 11  // 1011
val c = b.inv()
println(c)      // -12
```

Условные выражения

Условные выражения представляют некоторое условие, которое возвращает значение типа Boolean: либо true (если условие истинно), либо false (если условие ложно).

Операции отношения

- > (больше чем): возвращает true, если первый операнд больше второго. Иначе возвращает false

```
val a = 11
val b = 12
val c : Boolean =  a > b
println(c)      // false - a меньше чем b
```

```
val d = 35 > 12
println(d)      // true - 35 больше чем 12
```

- < (меньше чем): возвращает true, если первый операнд меньше второго. Иначе возвращает false

```
val a = 11
val b = 12
val c =  a < b   // true

val d = 35 < 12  // false
```

- >= (больше чем или равно): возвращает true, если первый операнд больше или равен второму

```
val a = 11
val b = 12
val c = a >= b      // false
val d = 11 >= a     // true
```

- <= (меньше чем или равно): возвращает true, если первый операнд меньше или равен второму.

```
val a = 11
val b = 12
val c = a <= b      // true
val d = 15 <= a     // false
```

- == (равно): возвращает true, если оба операнда равны. Иначе возвращает false

```
val a = 11
val b = 12
val c = a == b      // false
val d = b == 12     // true
```

- != (не равно): возвращает true, если оба операнда НЕ равны

```
val a = 11
val b = 12
val c = a != b      // true
val d = b != 12     // false
```

Логические операции

Операндами в логических операциях являются два значения типа Boolean. Нередко логические операции объединяют несколько операций отношения:

- and: возвращает true, если оба операнда равны true.

```
val a = true
val b = false
val c = a and b                          // false
val d = (11 >= 5) and (9 < 10)     // true
println(c)
println(d)
```

- or: возвращает true, если хотя бы один из операндов равен true.

```
val a = true
val b = false
val c = a or b                           // true
val d = (11 < 5) or (9 > 10)     // false
```

- xor: возвращает true, если только один из операндов равен true. Если операнды равны, возвращается false

```
val a = true
val b = false
val c = a xor b                 // true
val d = a xor (90 > 10)      // false
```

- !: возвращает true, если операнд равен false. И, наоборот, если операнд равен true, возвращается false.

```
val a = true
val b = !a  // false
val c = !b  // true
```

В качестве альтернативы оператору ! можно использовать метод not():

```
val a = true
val b = a.not()  // false
val c = b.not()  // true
```

- in: возвращает true, если операнд имеется в некоторой последовательности.

```
val a = 5
val b = a in 1..6       // true - число 5 входит в последовательность от 1 до 6

val c = 4
val d = c in 11..15     // false - число 4 НЕ входит в последовательность от 11 до
15
```

Выражение 1..6 создает последовательность чисел от 1 до 6. И в данном случае оператор in проверяет, есть ли значение переменной a в этой последовательности. Поскольку значение переменной a имеется в данной последовательности, то возвращается true.

А выражение 11..15 создает последовательность чисел от 11 до 15. И поскольку значение переменной c в эту последовательность не входит, поэтому возвращается false.

Если нам, наоборот, хочется возвращать true, если числа нет в указанной последовательности, то можно применить комбинацию операторов !in:

```
val a = 8
val b = a !in 1..6      // true - число 8 не входит в последовательность от 1 до 6
```

Условные конструкции

Условные конструкции позволяют направить выполнение программы по одному из путей в зависимости от условия.

if...else Конструкция if принимает условие, и если это условие истинно, то выполняется последующий блок инструкций.

```
val a = 10
if(a == 10) {

    println("a равно 10")
}
```

В данном случае в конструкции if проверяется истинность выражения a == 10, если оно истинно, то выполняется последующий блок кода в фигурных скобках, и на консоль выводится сообщение "a равно 10". Если же выражение ложно, тогда блок кода не выполняется.

Если необходимо задать альтернативный вариант, то можно добавить блок else:

```
val a = 10
if(a == 10) {
    println("a равно 10")
}
```

```
else{
    println("a НЕ равно 10")
}
```

Таким образом, если условное выражение после оператора if истинно, то выполняется блок после if, если ложно - выполняется блок после else.

Если блок кода состоит из одного выражения, то в принципе фигурные скобки можно опустить:

```
val a = 10
if(a == 10)
    println("a равно 10")
else
    println("a НЕ равно 10")
```

Если необходимо проверить несколько альтернативных вариантов, то можно добавить выражения else if:

```
val a = 10
if(a == 10) {
    println("a равно 10")
}
else if(a == 9){
    println("a равно 9")
}
else if(a == 8){
    println("a равно 8")
}
else{
    println("a имеет неопределенное значение")
}
```

Возвращение значения из if

Стоит отметить, что конструкция if может возвращать значение. Например, найдем максимальное из двух чисел:

```
val a = 10
val b = 20
val c = if (a > b) a else b

println(c)  // 20
```

Если при определении возвращаемого значения надо выполнить еще какие-нибудь действия, то можно заключить эти действия в блоки кода:

```
val a = 10
val b = 20
val c = if (a > b){
    println("a = $a")
    a
} else {
    println("b = $b")
    b
}
```

В конце каждого блока указывается возвращаемое значение.

Конструкция when

Конструкция when проверяет значение некоторого объекта и в зависимости от его значения выполняет тот или иной код. Конструкция when аналогична конструкции switch в других языках. Формальное определение:

```
when(объект){

    значение1 -> действия1
    значение2 -> действия2

    значениеN -> действияN
}
```

Если значение объекта равно одному из значений в блоке кода when, то выполняются соответствующие действия, которые идут после оператора -> после соответствующего значения.

Например:

```
fun main() {

    val isEnabled = true
    when(isEnabled){
        false -> println("isEnabled off")
        true -> println("isEnabled on")
    }
}
```

Здесь в качестве объекта в конструкцию when передается переменная isEnabled. Далее ее значение по порядку сравнивается со значениями в false и true. В данном случае переменная isEnabled равна true, поэтому будет выполняться код

```
println("isEnabled on")
```

Выражение else

В примере выше переменная isEnabled имела только два возможных варианта: true и false. Однако чаще бывают случаи, когда значения в блоке when не покрывают все возможные значения объекта. Дополнительное выражение else позволяет задать действия, которые выполняются, если объект не соответствует ни одному из значений. Например:

```
val a = 30
when(a){
    10 -> println("a = 10")
    20 -> println("a = 20")
    else -> println("неопределенное значение")
}
```

То есть в данном случае если переменная a равна 30, поэтому она не соответствует ни одному из значений в блоке when. И соответственно будут выполняться инструкции из выражения else.

Если надо, чтобы при совпадении значений выполнялось несколько инструкций, то для каждого значения можно определить блок кода:

```
var a = 10
when(a){
    10 -> {
        println("a = 10")
        a *= 2
    }
    20 -> {
        println("a = 20")
        a *= 5
    }
    else -> { println("неопределенное значение")}
}
println(a)
```

Сравнение с набором значений

Можно определить одни и те же действия сразу для нескольких значений. В этом случае значения перечисляются через запятую:

```
val a = 10
when(a){
    10, 20 -> println("a = 10 или a = 20")
```

```
    else -> println("неопределенное значение")
}
```

Также можно сравнивать с целым диапазоном значений с помощью оператора in:

```
val a = 10
when(a){
    in 10..19 -> println("a в диапазоне от 10 до 19")
    in 20..29 -> println("a в диапазоне от 20 до 29")
    !in 10..20 -> println("a вне диапазона от 10 до 20")
    else -> println("неопределенное значение")
}
```

Если оператор in позволяет узнать, есть ли значение в определенном диапазоне, то связка операторов !in позволяет проверить отсутствие значения в определенной последовательности.

when и динамически вычисляемые значения

Выражение в when также может сравниваться с динамически вычисляемыми значениями:

```
fun main() {

    val a = 10
    val b = 5
    val c = 3
    when(a){
        b - c -> println("a = b - c")
        b + 5 -> println("a = b + 5")
        else -> println("неопределенное значение")
    }
}
```

Так, в данном случае значение переменной a сравнивается с результатом операций b - c и b + 5.

Кроме того, when также может может принимать динамически вычисляемый объект:

```
fun main() {

    val a = 10
    val b = 20
    when(a + b){
        10 -> println("a + b = 10")
        20 -> println("a + b = 20")
        30 -> println("a + b = 30")
        else -> println("Undefined")
    }
}
```

Можно даже определять переменные, которые будут доступны внутри блока when:

```
fun main() {

    val a = 10
    val b = 26
    when(val c = a + b){
        10 -> println("a + b = 10")
        20 -> println("a + b = 20")
        else -> println("c = $c")
    }
}
```

when как альтернатива для if..else

Причем аналогично конструкции if..else просто может поверять набор условий и если одно из условий возвращает true, то выполнять соответствующий набор действий:

```
fun main() {

    val a = 15
    val b = 6
    when{
        (b > 10) -> println("b больше 10")
        (a > 10) -> println("a больше 10")
        else -> println("и a, и b меньше или равны 10")
    }
}
```

Возвращение значения

Как и if конструкция when может возвращать значение. Возвращаемое значение указывается после оператора ->:

```
val sum = 1000

val rate = when(sum){
    in 100..999 -> 10
    in 1000..9999 -> 15
    else -> 20
}
println(rate)       // 15
```

Таким образом, если значение переменной sum располагается в определенном диапазоне, то возвращается то значение, которое идет после стрелки.

Циклы

Циклы представляют вид управляющих конструкций, которые позволяют в зависимости от определенных условий выполнять некоторое действие множество раз.

For

Цикл for пробегается по всем элементам коллекции. В этом плане цикл for в Kotlin эквивалентен циклу for-each в ряде других языков программирования. Его формальная форма выглядит следующим образом:

```
for(переменная in последовательность){
    выполняемые инструкции
}
```

Например, выведем все квадраты чисел от 1 до 9, используя цикл for:

```
for(n in 1..9){
    print("${n * n} \t")
}
```

В данном случае перебирается последовательность чисел от 1 до 9. При каждом проходе цикла (итерации цикла) из этой последовательности будет извлекаться элемент и помещаться в переменную n. И через переменную n можно манипулировать значением элемента. То есть в данном случае мы получим следующий консольный вывод:

```
1   4   9   16  25  36  49  64  81
```

Циклы могут быть вложенными. Например, выведем таблицу умножения:

```
for(i in 1..9){
    for(j in 1..9){
        print("${i * j} \t")
    }
    println()
}
```

В итоге на консоль будет выведена следующая таблица умножения:

```
1   2   3   4   5   6   7   8   9
2   4   6   8   10  12  14  16  18
3   6   9   12  15  18  21  24  27
```

```
4   8   12  16  20  24  28  32  36
5   10  15  20  25  30  35  40  45
6   12  18  24  30  36  42  48  54
7   14  21  28  35  42  49  56  63
8   16  24  32  40  48  56  64  72
9   18  27  36  45  54  63  72  81
```

Цикл while

Цикл while повторяет определенные действия пока истинно некоторое условие:

```
var i = 10
while(i > 0){
    println(i*i)
    i--;
}
```

Здесь пока переменная i больше 0, будет выполняться цикл, в котором на консоль будет выводиться квадрат значения i.

В данном случае вначале проверяется условие (i > 0) и если оно истинно (то есть возвращает true), то выполняется цикл. И вполне может быть ситуация, когда к началу выполнения цикла условие не будет выполняться. Например, переменная i изначально меньше 0, тогда цикл вообще не будет выполняться.

Но есть и другая форма цикла while - do..while:

```
var i = -1
do{
    println(i*i)
    i--;
}
while(i > 0)
```

В данном случае вначале выполняется блок кода после ключевого слова do, а потом оценивается условие после while. Если условие истинно, то повторяется выполнение блока после do. То есть несмотря на то, что в данном случае переменная i меньше 0 и она не соответствует условию, тем не менее блок do выполнится хотя бы один раз.

Операторы continue и break

Иногда при использовании цикла возникает необходимость при некоторых условиях не дожидаться выполнения всех инструкций в цикле, перейти к новой итерации. Для этого можно использовать оператор continue:

```
for(n in 1..8){
    if(n == 5) continue;
```

```
    println(n * n)
}
```

В данном случае когда n будет равно 5, сработает оператор continue. И последующая инструкция, которая выводит на консоль квадрат числа, не будет выполняться. Цикл перейдет к обработке следующего элемента в массиве

Бывает, что при некоторых условиях нам вовсе надо выйти из цикла, прекратить его выполнение. В этом случае применяется оператор break:

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

В данном случае когда n окажется равен 5, то с помощью оператора break будет выполнен выход из цикла. Цикл полностью завершится.

Диапазоны

Диапазон представляет набор значений или неокторый интервал. Для создания диапазона применяется оператор ..:

```
val range = 1..5    // диапазон [1, 2, 3, 4, 5]
```

Этот оператор принимает два значения - границы диапазона, и все элементы между этими значениями (включая их самих) составляют диапазон.

Диапазон необязательно должна представлять числовые данные. Например, это могут быть строки:

```
val range =  "a".."d"
```

Оператор .. позволяет создать диапазон по нарастающей, где каждый следующий элемент будет больше предыдущего. С помощью специальной функции downTo можно построить диапазон в обратном порядке:

```
val range1 =  1..5      // 1 2 3 4 5
val range2 =  5 downTo 1    // 5 4 3 2 1
```

Еще одна специальная функция step позволяет задать шаг, на который будут изменяться последующие элементы:

```
val range1 = 1..10 step 2           // 1 3 5 7 9
val range2 = 10 downTo 1 step 3     // 10 7 4 1
```

Еще одна функция until позволяет не включать верхнюю границу в диапазон:

```
val range1 = 1 until 9          // 1 2 3 4 5 6 7 8
val range2 = 1 until 9 step 2   // 1 3 5 7
```

С помощью специальных операторов можно проверить наличие или отсутствие элементов в диапазоне:

- in: возвращает true, если объект имеется в диапазоне
- !in: возвращает true, если объект отсутствует в диапазоне

```
fun main() {

    val range = 1..5

    var isInRange = 5 in range
    println(isInRange)      // true

    isInRange = 86 in range
    println(isInRange)      // false

    var isNotInRange = 6 !in range
    println(isNotInRange)   // true

    isNotInRange = 3 !in range
    println(isNotInRange)   // false
}
```

Перебор диапазона

С помощью цикла for можно перебирать диапазон:

```
val range1 = 5 downTo 1
for(c in range1) print(c)   // 54321
println()

val range2 = 'a'..'d'
for(c in range2) print(c)   // abcd
println()

for(c in 1..9) print(c)     // 123456789
println()
```

```
for(c in 1 until 9) print(c)    // 12345678
println()

for(c in 1..9 step 2) print(c)  // 13579
```

Массивы

Для хранения набора значений в Kotlin, как и в других языках программирования, можно использовать массивы. При этом массив может хранить данные только одного того же типа. В Kotlin массивы представлены типом Array.

При определении массива после типа Array в угловых скобках необходимо указать, объекты какого типа могут храниться в массиве. Например, определим массив целых чисел:

```
val numbers: Array<Int>
```

С помощью встроенной функции arrayOf() можно передать набор значений, которые будут составлять массив:

```
val numbers: Array<Int> = arrayOf(1, 2, 3, 4, 5)
```

То есть в данном случае в массиве 5 чисел от 1 до 5.

С помощью индексов мы можем обратиться к определенному элементу в массиве. Индексация начинается с нуля, то есть первый элемент будет иметь индекс 0. Индекс указывается в квадратных скобках:

```
val numbers: Array<Int> = arrayOf(1, 2, 3, 4, 5)
val n = numbers[1]  // получаем второй элемент  n=2
numbers[2] = 7      // переустанавливаем третий элемент
println("numbers[2] = ${numbers[2]}") // numbers[2] = 7
```

Также инициализировать массив значениями можно следующим способом:

```
val numbers = Array(3, {5}) // [5, 5, 5]
```

Здесь применяется конструктор класса Array. В этот конструктор передаются два параметра. Первый параметр указывает, сколько элементов будет в массиве. В данном случае 3 элемента. Второй параметр представляет выражение, которое генерирует элементы массива. Оно заключается в фигурные скобки.

В данном случае в фигурных скобках стоит число 5, то есть все элементы массива будут представлять число 5. Таким образом, массив будет состоять из трех пятерок.

Но выражение, которое создает элементы массива, может быть и более сложным. Например:

```
var i = 1;
val numbers = Array(3, { i++ * 2}) // [2, 4, 6]
```

В данном случае элемент массива является результатом умножения переменной i на 2. При этом при каждом обращении к переменой i ее значение увеличивается на единицу.

Перебор массивов

Для перебора массивов можно применять цикл for:

```
fun main() {

    val numbers = arrayOf(1, 2, 3, 4, 5)
    for(number in numbers){
        print("$number \t")
    }
}
```

В данном случае переменная numbers представляет массив чисел. При переборе этого массива в цикле каждый его элемент оказывается в переменной number, значение которой, к примеру, можно вывести на консоль. Консольный вывод программы:

```
1   2   3   4   5
```

Подобным образом можно перебирать массивы и других типов:

```
fun main() {

    val people = arrayOf("Tom", "Sam", "Bob")
    for(person in people){
        print("$person \t")
    }
}
```

Консольный вывод программы:

```
Tom    Sam    Bob
```

Можно применять и другие типы циклов для перебора массива. Например, используем цикл while:

```
fun main() {

    val people = arrayOf("Tom", "Sam", "Bob")

    var i = 0
    while( i in people.indices){
        println(people[i])
        i++;
    }
}
```

Здесь определена дополнительная переменная i, которая представляет индекс элемента массива. У массива есть специальное свойство indices, которое содержит набор всех индексов. А выражение i in people.indices возвращает true, если значение переменной i входит в набор индексов массива.

В самом цикле по индексу обращаемся к элементу массива: println(people[i]). И затем переходим к следующему индексу, увеличивая счетчик: i++.

То же самое мы могли написать с помощью цикла for:

```
for (i in people.indices) {
    println(people[i])
}
```

Проверка наличия элемента в массиве

Как и в случае с последовательностью мы можем проверить наличие или отсутствие элементов в массиве с помощью операторов in и !in:

```
val numbers: Array<Int> = arrayOf(1, 2, 3, 4, 5)

println(4 in numbers)       // true
println(2 !in numbers)      // false
```

Массивы для базовых типов

Для упрощения создания массива в Kotlin определены дополнительные типы BooleanArray, ByteArray, ShortArray, IntArray, LongArray, CharArray, FloatArray и DoubleArray, которые позволяют создавать массивы для определенных типов. Например, тип IntArray позволяет определить массив объектов Int, а DoubleArray - массив объектов Double:

```
val numbers: IntArray = intArrayOf(1, 2, 3, 4, 5)
val doubles: DoubleArray = doubleArrayOf(2.4, 4.5, 1.2)
```

Для определения данных для этих массивов можно применять функции, которые начинаются на название типа в нижнем регистре, например, int, и затем идет ArrayOf.

Аналогично для инициализации подобных массивов также можно применять конструктор соответствующего класса:

```
val numbers = IntArray(3, {5})
val doubles = DoubleArray(3, {1.5})
```

Двухмерные массивы

Выше рассматривались одномерные массивы, которые можно представить в виде ряда или строки значений. Но кроме того, мы можем использовать многомерные массивы. К примеру, возьмем двухмерный массив - то есть такой массив, каждый элемент которого в свою очередь сам является массивом. Двухмерный массив еще можно представить в виде таблицы, где каждая строка - это отдельный массив, а ячейки строки - это элементы вложенного массива.

Определение двухмерных массивов менее интуитивно понятно и может вызывать сложности. Например, двухмерный массив чисел:

```
val table: Array<Array<Int>> = Array(3, { Array(5, {0}) })
```

В данном случае двухмерный массив будет иметь три элемента - три строки. Каждая строка будет иметь по пять элементов, каждый из которых равен 0.

Используя индексы, можно обращаться к подмассивам в подобном массиве, в том числе переустанавливать их значения:

```
val table = Array(3, { Array(3, {0}) })
table[0] = arrayOf(1, 2, 3)     // первая строка таблицы
table[1] = arrayOf(4, 5, 6)     // вторая строка таблицы
table[2] = arrayOf(7, 8, 9)     // третья строка таблицы
```

Для обращения к элементам подмассивов двухмерного массива необходимы два индекса. По первому индексу идет получение строки, а по второму индексу - столбца в рамках этой строки:

```
val table = Array(3, { Array(3, {0}) })
table[0][1] = 6  // второй элемент первой строки
val n = table[0][1]     // n = 6
```

Используя два цикла, можно перебирать двухмерные массивы:

```
fun main() {

    val table: Array<Array<Int>> = Array(3, { Array(3, {0}) })
    table[0] = arrayOf(1, 2, 3)
    table[1] = arrayOf(4, 5, 6)
    table[2] = arrayOf(7, 8, 9)
    for(row in table){

        for(cell in row){
            print("$cell \t")
        }
        println()
    }
}
```

С помощью внешнего цикла for(row in table) пробегаемся по всем элементам двухмерного массива, то есть по строкам таблицы. Каждый из элементов двухмерного массива сам представляет массив, поэтому мы можем пробежаться по этому массиву и получить из него непосредственно те значения, которые в нем хранятся. В итоге на консоль будет выведено следующее:

```
1   2   3
4   5   6
7   8   9
```