



## Лекция #4. Типы данных и операции в Kotlin, часть 1

### Преобразование типов данных в Kotlin

Иногда данные из одного формата требуется преобразовать в другой. Обратите внимание, что попытка выполнить конвертацию типа следующим образом обернется ошибкой:

```
1 var integer: Int = 100
2 var decimal: Double = 12.5
3 integer = decimal
```

Текст ошибки, которую выдает Kotlin при попытке выполнить преобразование таким образом: **Type mismatch: inferred type is Double but Int was expected**

Некоторые языки программирования менее строгие и выполняют конвертацию подобного рода автоматически. Опыт показывает, что такое автоматическое преобразование является источником ошибок в программах и зачастую снижает производительность.

**Kotlin** запрещает присваивать значение одного типа другому и тем самым помогает избежать проблем.

Помните, что именно программисты заставляют компьютеры работать. В **Kotlin** частью задачи разработчика является явное преобразование типов. Если вам нужно преобразование, вы должны сообщить программе об этом сами.

Вместо простого присваивания требуется явно сообщить о необходимости преобразования типа. Это делается следующим образом:

```
1 | integer = decimal.toInt()
```

Теперь присваивание однозначно сообщает **Kotlin**, что вы хотите преобразовать исходный тип **Double** в новый тип **Int**.

На заметку: В данном случае присваивание десятичного значения целому числу приводит к потере точности: переменная **integer** получает значение 12 вместо 12.5. Именно по этой причине важно работать явно. **Kotlin** хочет убедиться, что вы знаете, что делаете, и что вы можете потерять данные, выполняя преобразование типов.

## Операции со смешанными типами в Kotlin

До сих пор вы видели только операторы, действующие независимо от целых или десятичных чисел. Но что, если у вас есть целое число, которое требуется умножить на десятичное?

У вас может возникнуть идея сделать что-то вроде этого:

```
1 | val hourlyRate: Double = 19.5
2 | val hoursWorked: Int = 10
3 | val totalCost: Double = hourlyRate * hoursWorked.toDouble()
```

В данном примере константа **hoursWorked** явно конвертируется в тип **Double** для совпадения с типом константы **hourlyRate**. Однако, делать это нет нужды. **Kotlin** позволяет умножать эти значения без всякой конвертации следующим образом:

```
1 | val totalCost: Double = hourlyRate * hoursWorked
```

В **Kotlin** оператор **\*** можно использовать со смешанным типом. Это правило также применяется к другим арифметическим операторам. Несмотря на то, что **hoursWorked** является типом **Int**, это не повлияет на точность **hourlyRate**. Результатом все равно будет **195.0**.

**Kotlin** старается быть как можно более кратким и пытается вывести ожидаемое, когда это возможно. Таким образом, вы тратите меньше времени на заботу о типах и больше на создание кода.

## Вывод типов данных в Kotlin (Type inference)

До сих пор каждое объявление переменной или константы сопровождалась объявлением типа. Может возникнуть вопрос, зачем писать **: Int** и **: Double**, ведь правая часть присваивания уже является **Int** или **Double**. Разумеется, это излишне, это ведь и так понятно.

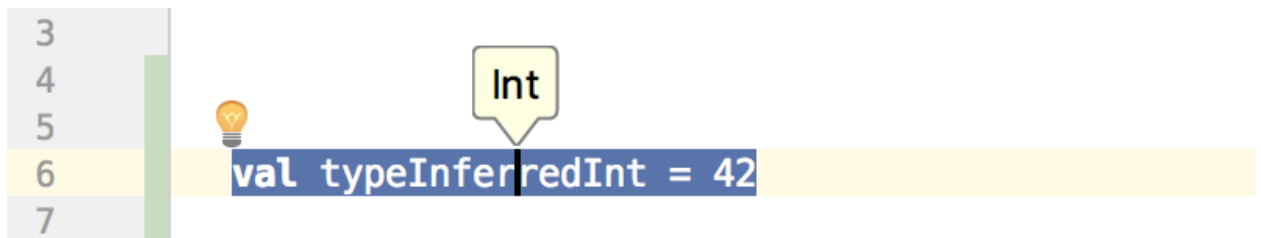
Оказывается, компилятору **Kotlin** это тоже понятно. Ему не нужно постоянно указывать тип — он может определить его самостоятельно. Это делается с помощью процесса, который называется выводом типа. Это ключевой компонент **Kotlin**, который есть далеко не в каждом языке программирования.

Таким образом, вы можете просто не указывать тип в большинстве мест, где вы его видите.

К примеру, рассмотрим следующее объявление константы:

```
1 | val typeInferredInt = 42
```

Иногда будет полезно проверить предполагаемый тип переменной или константы. Вы можете сделать это в **IntelliJ**, нажав на название переменной или константы и удерживая клавиши **Control + Shift + P**. **IntelliJ** отобразит всплывающее окно, наподобие следующего:

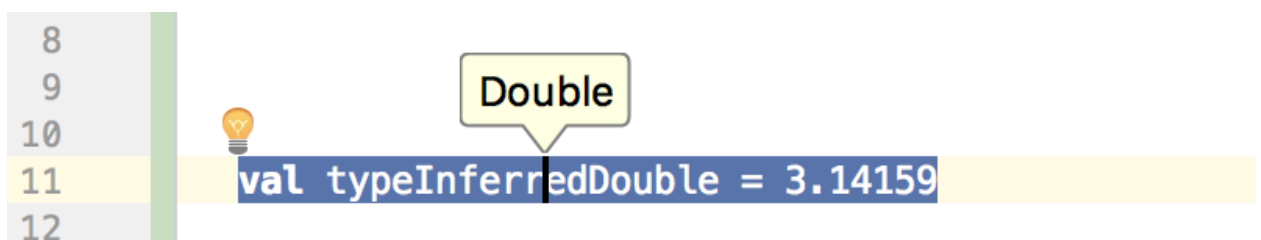


**IntelliJ** показывает предполагаемый тип, предоставляя вам объявление, которое вам пришлось бы использовать, если бы не было вывода типа (**type inference**). В данном случае это тип **Int**.

Для других типов это также работает:

```
1 | val typeInferredDouble = 3.14159
```

При удерживании клавиш **Control + Shift + P** будет показано следующее:



Из данных примеров видно, что в выводе типов никакой магии нет. **Kotlin** просто делает то же самое, что и ваш мозг. Языки программирования без вывода типов часто могут показаться многословными, потому что зачастую нужно указывать очевидный тип каждый раз при объявлении переменной и константы.

На заметку: В дальнейшем вы узнаете о более сложных типах данных, для которых **Kotlin** не всегда может определить тип автоматически. Однако это довольно редкий случай.

В некоторых случаях требуется определить константу или переменную и убедиться в их типе, даже если вы назначаете другой тип. Ранее вы видели, как можно преобразовать один тип в другой. Например, рассмотрим следующий пример:

```
1 | val wantADouble = 3
```

Здесь **Kotlin** выберет **Int** в качестве типа константы **wantADouble**. Однако, что если вам нужен вместо **Int** тип **Double**?

Сначала требуется сделать следующее:

```
1 | val actuallyDouble = 3.toDouble()
```

Это похоже на преобразование типа, которое вы видели ранее.

Другой опцией стал бы отказ от использования вывода типов вообще:

```
1 | val actuallyDouble: Double = 3.0
```

Быть может, вам захочется сделать следующее:

```
1 | val actuallyDouble: Double = 3
```

Этого делать нельзя, компилятор выведет следующую ошибку: **The integer literal does not conform to the expected type Double**.

На заметку: В **Kotlin** литеральные значения, такие как 3, имеют определенный тип — **Int**. Если вы хотите преобразовать их в другой тип, вы должны сделать это явно, например, вызвав **toDouble()**. Литеральное числовое значение, содержащее десятичную точку, может использоваться только как **Double** и должно быть явно преобразовано, если вы хотите использовать его как что-то еще. По этой причине нельзя присвоить значение 3 константе **actualDouble**.

Точно так же литеральные числовые значения, которые содержат десятичную точку, не могут быть целыми числами **integer**. Это означает, что вы могли бы избежать всего этого обсуждения, если бы написали так:

```
val wantADouble = 3.0
```

## Строки в Kotlin

Числа лежат в основе программирования, но вы будете работать не только с ними. Текстовые данные также является очень распространенным типом. Это могут быть имена людей, адреса или даже книги. Все это примеры текста, который может потребоваться обработать приложению.

Большинство языков программирования хранит текст в виде типа данных, который называется строкой, или **string**.

### Как компьютер воспринимает строки

Компьютеры воспринимают строки как набор отдельных символов. Ранее вы узнали, что числа представляют собой язык процессоров, и любые языки программирования можно свести к необработанным числам. Строки ничем не отличаются.

Это может показаться очень странным. Как символы могут быть числами? Компьютер умеет переводить символ на собственный язык. Он это делает, присваивая каждому символу другой номер. Это формирует двустороннее отображение символа в число, которое называется набором символов.

При нажатии клавиши символа на клавиатуре вы передаете компьютеру номер символа. Текстовый редактор преобразует число в изображение символа и показывает вам это изображение.

### Unicode

Изолированно, компьютер может выбирать любое сопоставление набора символов, которое ему нравится. Если нужно, чтобы буква а была равна числу 10, пусть так и будет. Но когда компьютеры начинают общаться друг с другом, им необходимо использовать общий набор символов. Ведь при задействовании разных наборов символов возникает путаница, так как компьютеры рассматривают разные образцы символов.

На протяжении многих лет существовало несколько стандартов, но самым современным стандартом является **Unicode**. Он определяет отображение набора символов, которое сегодня используют почти все компьютеры.

В качестве примера рассмотрим слово `safe`. Стандарт **Unicode** указывает, что буквы данного слова нужно представить следующим образом:



c	a	f	e
99	97	102	101

Число, связанное с каждым символом, называется кодовой точкой. В приведенном выше примере **c** использует кодовую точку **99**, **a** использует кодовую точку **97** и так далее.

Конечно, **Unicode** предназначен не только для простых латинских символов, используемых в английском языке вроде c, a, f и e. Он также позволяет отображать символы других языков. Слово *café* происходит от французского, в котором оно пишется как *café*. Unicode отображает эти символы следующим образом:

c	a	f	é
99	97	102	233

Вы наверняка не раз использовали смайлики, или эмодзи. По сути они являются символами, преобразованными с помощью **Unicode**. К примеру:

	
128169	128512

Это только два символа. Кодовыми точками данных символов являются довольно крупные числа, но каждое из них представляет собой только одну

кодovou точку. Компьютер воспринимает такие смайлики точно так же, как и любые другие символы.

## Char и String — Символы и строки в Kotlin

**Kotlin**, как и любой современный язык программирования, может работать напрямую с символами и строками. Для этого используются типы **Char** и **String**. Далее мы подробно рассмотрим данные типы и то, как с ними работать.

Тип данных **Char** может хранить один символ, который должен быть обрaмлен в одинарные кавычки. К примеру:

```
1 | val characterA: Char = 'a'
```

Этот тип данных предназначен для хранения только одного символа. С другой стороны, тип **String** хранит много символов, которые должны помещаться внутри двойных кавычек. К примеру:

```
1 | val stringDog: String = "Dog"
```

Все просто. Правая часть данного выражения представляет собой строковый литерал. Это синтаксис **Kotlin** для представления строки.

Здесь также применяется вывод типа. Если вы удалите тип в приведенном выше объявлении переменной, **Kotlin** сам поймет, что **stringDog** является константой типа **String**:

```
1 | val stringDog = "Dog" // Вывод типа String
```

## Конкатенация строк в Kotlin

После создания строк вы можете совершить над ними различные действия. Одним из самых популярных способов манипуляции со строкой является ее объединение с другой строкой.

В **Kotlin** это делается довольно просто: с помощью оператора сложения **+**. Строки складываются точно так же, как числа:

```
1 | var message = "Привет!" + " Меня зовут "  
2 | val name = "Андрей"  
3 | message += name // Вывод: "Привет! Меня зовут Андрей"
```

Вам нужно объявить переменную **message**, а не константу, потому что ее нужно будет модифицировать. Можно складывать строковые литералы вместе,

как в первой строке, и можно складывать строковые переменные или константы вместе, как в последней строке.

Также возможно добавлять символы прямо в строку. Это похоже на работу с числами, если одна переменная типа **Int**, а другое — **Double**.

Для добавления символа в строку сделайте следующее:

```
1 | val exclamationMark: Char = '!'
2 | message += exclamationMark // Вывод: "Привет! Меня зовут Андрей!"
```

Нет нужды конвертировать **Character** в **String** перед добавлением в **message**, **Kotlin** сделает это сам.

## Шаблонные строки в Kotlin

В **Kotlin** можно создавать строки, используя шаблоны. Они используют специальный синтаксис, который позволяет создать строку, которая легко читается:

```
1 | message = "Привет! Меня зовут $name!" // Вывод: "Привет! Меня зовут Андрей!"
```

Это читается удобнее, чем пример из предыдущего раздела. Это расширение синтаксиса строкового литерала, с помощью которого заменяются определенные части строки другими значениями. Просто добавьте перед значением, которое требуется вставить, символ **\$**.

Этот синтаксис используется и для создания строк из других типов данных, к примеру, чисел:

```
1 | val oneThird = 1.0 / 3.0
2 | val oneThirdLongString = "Одна треть равна $oneThird в виде десятичной дроби."
```

В шаблоне используется **Double**. В конце, константа **oneThirdLongString** будет содержать следующее:

```
1 | Одна треть равна 0.3333333333333333 в виде десятичной дроби.
```

Для представления одной трети десятичной дроби потребовалось бы бесконечное количество символов, потому что это периодическая десятичная дробь. Использование строковых шаблонов с **Double** не позволяет управлять точностью.

Это неприятные последствия использования строковых шаблонов — они просты, но не дают возможности настраивать вывод.



Внутри строкового шаблона также можно поместить выражения, поставив за символом **\$** пару фигурных скобок, которые содержат выражение:

```
1 | val oneThirdLongString = "Одна треть равна ${1.0 / 3.0} в виде десятичной дроби."
```

Результат будет прежним.

## Многострочный текст в Kotlin

В **Kotlin** есть удобный способ записи многострочного текста. Он может быть полезен при необходимости размещения очень длинной строки.

Это делается следующим образом:

```
1 | val bigString = ""  
2 | |Таким образом  
3 | |вы можете получить строку,  
4 | |содержащую  
5 | |несколько  
6 | |строк.  
7 | |"".trimMargin()  
8 |  
9 | println(bigString)
```

Три двойные кавычки указывают на многострочный текст. Первая и последняя строки не становятся частью содержимого переменной или константы. Это удобно, так как вам не нужно размещать три двойные кавычки в одной строке с содержимым. Что-то похожее мы делали при изучении многострочных комментариев.

Вывод кода будет следующим:

```
1 | Таким образом  
2 | вы можете получить строку,  
3 | содержащую  
4 | несколько  
5 | строк.
```

Обратите внимание на вертикальную черту **|** в начале строк и на вызов функции **trimMargin()**. Такая запись предотвращает появление в строке начальных пробелов, позволяя форматировать код с красивыми отступами, не влияя на вывод.

## Типы Pair и Triple в Kotlin

Иногда данные используются в группах. Примером может быть пара координат (x, y) на 2D сетке. Аналогично набор координат на 3D сетке состоит из значений x, y и z. В **Kotlin** для таких данных используются типы **Pair** и **Triple**.

В других языках похожую функцию выполняет тип кортежей.

### Типы Pair и Triple в Kotlin

**Pair** и **Triple** являются типами, которые представляют данные, состоящие из трех или двух значений любого типа. Если вам нужно больше трех значений, требуется использовать **data class**, который мы рассмотрим в будущем.

Рассмотрим тип **Pair**. В качестве примера определим пару 2D координат, где значение каждой оси является целым числом типа **Int**:

```
1 | val coordinates: Pair<Int, Int> = Pair(2, 3)
```

Типом константы **coordinates** является **Pair<Int, Int>**. Тип значений внутри **Pair**, в данном случае является **Int**, разделяются запятыми и заключаются в **<>**. Код для создания **Pair** почти такой же, каждое значение разделено запятыми и заключено в круглые скобки.

Вывод типа также может автоматически понять, что это тип **Pair**:

```
1 | val coordinatesInferred = Pair(2, 3)
```

Запись можно сделать более краткой, используя оператор **to**:

```
1 | val coordinatesWithTo = 2 to 3
```

Аналогичным образом в **Pair** можно поместить значения типа **Double**:

```
1 | val coordinatesDoubles = Pair(2.1, 3.5)
2 | // Вывод типа будет следующим: Pair<Double, Double>
```

Получить доступ к данным внутри **Pair** можно так:

```
1 | val x1 = coordinates.first
2 | val y1 = coordinates.second
```

Можно ссылаться на каждый элемент в **Pair** по его положению в паре, начиная с **first**. В этом примере **x1** будет равен 2, а **y1** будет равен 3.

В предыдущем примере не сразу понятно, что первое значение — это координата **x**, а второе значение — координата **y**. Это еще один пример того, почему при объявлении переменной важно давать переменным понятные названия.

**Kotlin** позволяет использовать объявление *деструктуризации* для отдельных частей **Pair**. Таким образом можно четко указать, что из себя представляет каждая часть. К примеру:

```
1 | val (x, y) = coordinates
2 | // x и y принадлежат типу Int
```

Здесь значения извлекаются из **coordinates** и присваиваются к **x** и **y**.

**Triple** работает так же, как и **Pair**, только вместо двух значений у нас теперь три.

Если вам нужно одновременно получить доступ к частям данных из **Triple**, как в примере выше, можно использовать более простой синтаксис:

```
1 | val coordinates3D = Triple(2, 3, 1)
2 | val (x3, y3, z3) = coordinates3D
```

Здесь объявляются три новые константы — **x3**, **y3** и **z3**, им присваивается каждая часть из **Triple**. Код является эквивалентом следующему:

```
1 | val coordinates3D = Triple(2, 3, 1)
2 | val x3 = coordinates3D.first
3 | val y3 = coordinates3D.second
4 | val z3 = coordinates3D.third
```

Если вам нужно проигнорировать определенный элемент из **Pair** или **Triple**, можно заменить соответствующую часть объявления подчеркиванием. К примеру, при выполнении 2D вычислений необходимо проигнорировать координату **z** из **coordinates3D**. Это делается следующим образом:

```
1 | val (x4, y4, _) = coordinates3D
```

Эта строка кода объявляет только **x4** и **y4**. С помощью специального символа **\_** игнорируется указанная часть.

## Числовые типы данных в Kotlin

Во многих языках на основе **C** вроде **Java** есть примитивные типы данных, которые занимают определенное количество байтов. К примеру, в **Java** 32-битное примитивное число — это **int**. Существует также объектная версия **int**,

известная как **Integer**. Может возникнуть вопрос, зачем нужны два типа, если хранят они один и тот же числовой тип?

Примитивы требуют меньше памяти. Это означает, что они лучше по производительности, но им все-таки не хватает некоторых функций от **Integer**. Хорошая новость — в **Kotlin** вам не нужно беспокоиться о том, использовать примитивный или объектный тип. **Kotlin** делает все сам, поэтому вам нужно использовать обычный **Int**.

Вы использовали **Int** для целых чисел, которые представлены через 32 бита. В **Kotlin** есть много числовых типов, для хранения которых нужно различное количество памяти. К примеру, для целых чисел можно использовать **Byte**, **Short** и **Long**. Для этих типов нужно 1, 2 и 8 байтов соответственно. Каждый из данных типов использует один бит для представления знака.

Далее дается обзор различных типов целых чисел и размер, нужной для них памяти, в байтах. В большинстве случаев используется **Int**.

Тип	Минимальное значение	Максимальное значение	Память
Byte	-128	127	1
Short	-32768	32767	2
Int	-2147483648	2147483647	4
Long	-9223372036854775807	9223372036854775806	8

Для дробных чисел мы использовали тип **Double**. В **Kotlin** также есть тип **Float**, у которого меньший диапазон точности, чем у **Double**, и которому нужно в два раза меньше памяти. Современные программы оптимизированы под **Double**, поэтому лучше использовать его.

Тип	Минимальное значение	Максимальное значение	Точность	Память
Float	-3.4028235E+38	3.4028235E+38	6 цифр	1
Double	-1.797693E+308	1.797693E+308	15 цифр	2

В большинстве случаев для представления чисел используются типы **Int** и **Double**. Однако время от времени вы будете сталкиваться и с другими типами данных. Предположим, нужно сложить типы **Short**, **Byte** и **Long**. Это можно сделать следующим образом:

```
1 val a: Short = 12
2 val b: Byte = 120
3 val c: Int = -100000
4
5 val answer = a + b + c // Ответ будет иметь тип Int
```

## Типы Any, Unit и Nothing в Kotlin

Тип **Any** можно назвать матерью всех прочих типов (кроме типов **Null**, которые мы рассмотрим в будущем). В **Kotlin** каждый тип, будь то **Int** или **String**, считается **Any**. Это напоминает тип **object** в **C#**, который является корнем всех типов, кроме примитивных данных.

К примеру, в **Kotlin** можно объявить литералы **Int** и **String** как **Any**:

```
1 | val anyNumber: Any = 42
2 | val anyString: Any = "42"
```

### Тип Unit в Kotlin (void)

**Unit** является специальным типом, который всегда представляет только одно значение: объект **Unit**. Он похож на тип **void** в **C#**, только он упрощает работу с обобщениями, которые будут рассмотрены в будущем. Каждая функция (думайте о функции как о фрагменте кода многократного использования), которая явно не возвращает тип, например **String**, возвращает **Unit** т.е. **void** если ассоциация с **C#** вам более близка.

К примеру, далее идет код функции, которая просто складывает  $2 + 2$  и как бы должна вывести результат, но на самом деле ничего не возвращает:

```
1 | fun add() {
2 |     val result = 2 + 2
3 |     println(result)
4 | }
```

Результатом функции будет тип **Unit**, так что функция выше аналогична следующей:

```
1 | fun add(): Unit {
2 |     val result = 2 + 2
3 |     println(result)
4 | }
```

### Тип Nothing в Kotlin

**Nothing** является типом, который полезен при объявлении функции, которая не только ничего не возвращает, но и не завершается.

Это может произойти, если функция либо заставляет программу полностью остановиться, генерируя исключение (ошибку) **Exception**, либо просто продолжается бесконечно без завершения (вечный цикл).

К примеру:

```
1 fun doNothingForever(): Nothing {  
2     while(true) {  
3  
4     }  
5 }
```

В будущем мы подробнее рассмотрим цикл **while**, но пока достаточно понять, что данная функция продолжается бесконечно и ничего не возвращает.