

## Лекция. Использование типов функций и лямбда-выражений в Kotlin

### Введение

В этой практической работе вы узнаете о типах функций, о том, как использовать типы функций, а также о синтаксисе, характерном для **лямбда-выражений**.

В Kotlin функции считаются конструкциями первого класса. Это означает, что к функциям можно относиться как к типу данных. Вы можете хранить функции в переменных, передавать их другим функциям в качестве аргументов и возвращать их из других функций.

Как и другие типы данных, которые можно выразить с помощью литеральных значений - например, тип `Int` для значения 10 или тип `String` для значения «Hello», - вы также можете объявлять литералы функций, которые называются **лямбда-выражениями** или сокращенно лямбдами. Лямбда-выражения широко используются при разработке Android и, в целом, в программировании на Kotlin.

### Необходимые условия

- Знакомство с программированием на Kotlin, включая функции, операторы `if/else` и нулевые возможности.

### Что вы узнаете

- Как определить функцию с помощью синтаксиса лямбда.
- Как хранить функции в переменных.
- Как передавать функции в качестве аргументов другим функциям.
- Как возвращать функции из других функций.
- Как использовать нулевые типы функций.
- Как сделать лямбда-выражения более краткими.
- Что такое функция высшего порядка.
- Как использовать функцию `repeat()`.

### Хранение функции в переменной

До сих пор вы узнали, как объявлять функции с помощью ключевого слова **fun**. Функция, объявленная с помощью ключевого слова `fun`, может быть вызвана, что приводит к выполнению кода в теле функции.

Будучи первоклассной конструкцией, функции также являются типами данных, поэтому вы можете хранить функции в переменных, передавать их в функции и возвращать из функций. Возможно, вы хотите иметь возможность изменять поведение части приложения во время выполнения или вложить составные функции для создания макетов, как это было на предыдущих работах. Все это становится возможным благодаря лямбда-выражениям.

Вы можете увидеть это в действии с помощью трюка или угощения, что означает традицию Хэллоуина во многих странах, во время которой дети, одетые в костюмы, ходят от двери к двери и спрашивают: «Кошелек или угощение», обычно в обмен на конфеты.

Храните функцию в переменной:

После функции `main()` определите функцию `trick()` без параметров и возвращаемого значения, которая печатает «No treats!». Синтаксис такой же, как и у других функций, которые вы видели в предыдущих работах.

```
fun main() {  
  
}  
  
fun trick() {  
    println("No treats!")  
}
```

В теле функции `main()` создайте переменную `trickFunction` и установите ее равной `trick`. Вы не включаете круглые скобки после `trick`, потому что хотите сохранить функцию в переменной, а не вызвать ее.

```
fun main() {  
    val trickFunction = trick  
}  
  
fun trick() {  
    println("No treats!")  
}
```

Запустите свой код. Он выдает ошибку, потому что компилятор Kotlin распознает `trick` как имя функции `trick()`, но ожидает, что вы вызовете функцию, а не присвоите ее переменной.

```
Function invocation 'trick()' expected
```

Вы попытались сохранить трюк в переменной `trickFunction`. Однако, чтобы сослаться на функцию как на значение, необходимо использовать оператор ссылки на функцию (`::`). Синтаксис показан на этом рисунке:

::

function name

Чтобы сослаться на функцию как на значение, переназначьте `trickFunction` на `::trick`.

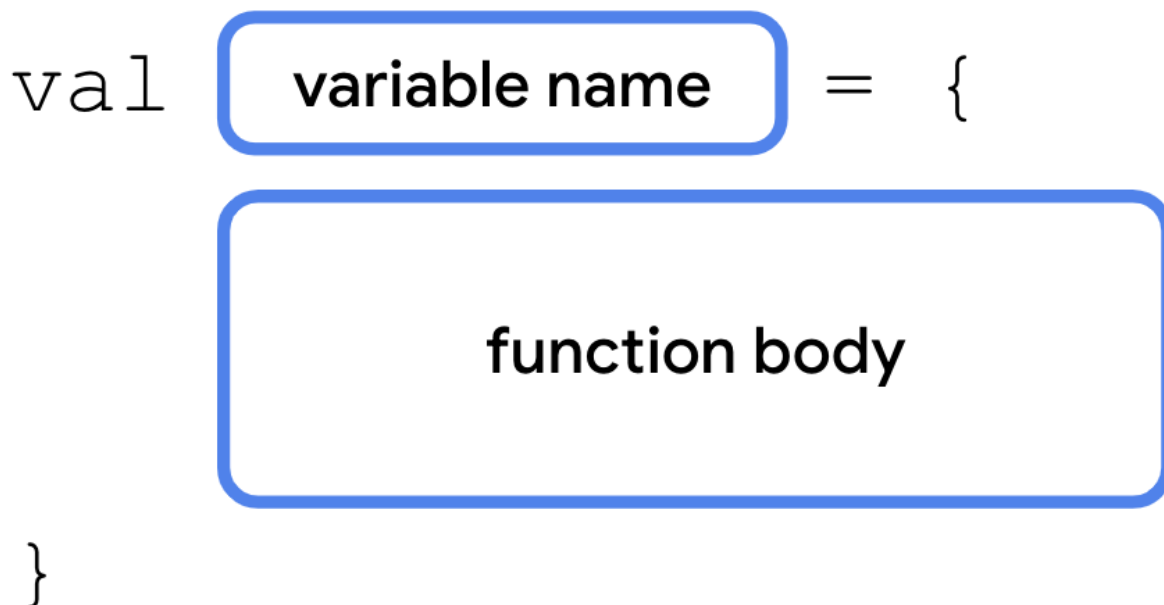
```
fun main() {  
    val trickFunction = ::trick  
}  
  
fun trick() {  
    println("No treats!")  
}
```

Запустите свой код, чтобы убедиться, что ошибок больше нет. Вы видите предупреждение о том, что функция `trickFunction` никогда не используется, но это будет исправлено в следующем разделе.

### Переопределите функцию с помощью лямбда-выражения

Лямбда-выражения обеспечивают лаконичный синтаксис для определения функции без ключевого слова `fun`. Вы можете хранить лямбда-выражение непосредственно в переменной без ссылки на другую функцию.

Перед оператором присваивания (`=`) добавляется ключевое слово `val` или `var`, за которым следует имя переменной, используемой при вызове функции. После оператора присваивания (`=`) идет лямбда-выражение, состоящее из пары фигурных скобок, которые образуют тело функции. Синтаксис показан на этом рисунке:



The diagram shows the syntax for a lambda expression assignment. It starts with the keyword `val`, followed by a rounded rectangle containing the text `variable name`. This is followed by an equals sign `=` and an opening curly brace `{`. Below the opening brace is a large rounded rectangle containing the text `function body`. The diagram ends with a closing curly brace `}`.

Когда вы определяете функцию с помощью лямбда-выражения, у вас появляется переменная, которая ссылается на функцию. Вы также можете присваивать ее значение другим переменным, как и любой другой тип, и вызывать функцию с именем новой переменной.

- Обновите код, чтобы использовать лямбда-выражение:

Перепишите функцию `trick()` с помощью лямбда-выражения. Теперь имя `trick` относится к имени переменной. Тело функции в фигурных скобках теперь представляет собой лямбда-выражение.

```
fun main() {  
    val trickFunction = ::trick  
}  
  
val trick = {  
    println("No treats!")  
}
```

В функции `main()` удалите оператор ссылки на функцию (`::`), потому что теперь `trick` ссылается на переменную, а не на имя функции.

```
fun main() {  
    val trickFunction = trick  
}  
  
val trick = {  
    println("No treats!")  
}
```

Запустите свой код. Ошибок нет, и вы можете ссылаться на функцию `trick()` без оператора ссылки на функцию (`::`). Вывода нет, потому что вы все еще не вызвали функцию.

- В функции `main()` вызовите функцию `trick()`, но на этот раз заключите ее в круглые скобки, как это делается при вызове любой другой функции.

```
fun main() {  
    val trickFunction = trick  
    trick()  
}  
  
val trick = {  
    println("No treats!")  
}
```

- Запустите свой код. Тело лямбда-выражения будет выполнено.

```
No treats!
```

В функции `main()` вызовите переменную `trickFunction`, как если бы это была функция.

```
fun main() {  
    val trickFunction = trick  
    trick()  
    trickFunction()  
}
```

```
}  
  
val trick = {  
    println("No treats!")  
}
```

- Запустите свой код. Функция вызывается дважды: один раз для вызова функции `trick()` и второй раз для вызова функции `trickFunction()`.

```
No treats!  
No treats!
```

С помощью лямбда-выражений вы можете создавать переменные, хранящие функции, вызывать эти переменные как функции и хранить их в других переменных, которые можно вызывать как функции.

## Использование функций в качестве типа данных

Вы узнали, что в Kotlin есть функция определения типов. Когда вы объявляете переменную, вам часто не нужно явно указывать ее тип. В предыдущем примере компилятор Kotlin смог сделать вывод, что значение `trick` - это функция. Однако если вы хотите указать тип параметра функции или возвращаемого типа, вам необходимо знать синтаксис для выражения типов функций.

Типы функций состоят из набора круглых скобок, которые содержат список необязательных параметров, символ `->` и возвращаемый тип. Синтаксис показан на этом рисунке:

( **parameters (optional)** ) `->` **return type**

Тип данных переменной `trick`, которую вы объявили ранее, будет `() -> Unit`. Круглые скобки пустые, потому что у функции нет параметров. Тип возвращаемых данных - `Unit`, потому что функция ничего не возвращает. Если бы у вас была функция, принимающая два параметра `Int` и возвращающая `Int`, то ее тип данных был бы `(Int, Int) -> Int`.

Объявите еще одну функцию с помощью лямбда-выражения, которое явно указывает тип функции:

- После переменной `trick` объявите переменную `treat`, равную лямбда-выражению с телом, которое печатает «Have a treat!».

```
val trick = {  
    println("No treats!")  
}  
  
val treat = {
```

```
println("Have a treat!")  
}
```

- Укажите тип данных переменной treat как () -> Unit.

```
val treat: () -> Unit = {  
    println("Have a treat!")  
}
```

В функции main() вызовите функцию treat().

```
fun main() {  
    val trickFunction = trick  
    trick()  
    trickFunction()  
    treat()  
}
```

- Запустите код. Функция treat() ведет себя так же, как и функция trick(). Обе переменные имеют одинаковый тип данных, хотя только переменная treat объявляет его явно.

```
No treats!  
No treats!  
Have a treat!
```

## Использование функции в качестве возвращаемого типа

Функция - это тип данных, поэтому вы можете использовать ее так же, как и любой другой тип данных. Вы даже можете возвращать функции из других функций. Синтаксис показан на этом рисунке:

```
fun function name ( ) : function type {  
    // code  
    return Name of another function  
}
```

Создайте функцию, которая возвращает функцию.

- Удалите код из функции main().

```
fun main() {  
  
}
```

- После функции main() определите функцию trickOrTreat(), которая принимает параметр isTrick типа Boolean.

```
fun main() {  
  
}  
  
fun trickOrTreat(isTrick: Boolean): () -> Unit {  
}  
  
val trick = {  
    println("No treats!")  
}  
  
val treat = {  
    println("Have a treat!")  
}
```

- В теле функции trickOrTreat() добавьте оператор if, который возвращает функцию trick(), если isTrick равен true, и возвращает функцию treat(), если isTrick равен false.

```
fun trickOrTreat(isTrick: Boolean): () -> Unit {  
    if (isTrick) {  
        return trick  
    } else {  
        return treat  
    }  
}
```

- В функции main() создайте переменную treatFunction и присвойте ее результату вызова функции trickOrTreat(), передав в качестве параметра isTrick значение false. Затем создайте вторую переменную под названием trickFunction и присвойте ее результату вызова функции trickOrTreat(), на этот раз передав true для параметра isTrick.

```
fun main() {  
    val treatFunction = trickOrTreat(false)  
    val trickFunction = trickOrTreat(true)  
}
```

- Вызовите функцию `treatFunction()`, а затем вызовите функцию `trickFunction()` в следующей строке.

```
fun main() {  
    val treatFunction = trickOrTreat(false)  
    val trickFunction = trickOrTreat(true)  
    treatFunction()  
    trickFunction()  
}
```

- Запустите свой код. Вы должны увидеть вывод для каждой функции. Даже если вы не вызывали функции `trick()` или `treat()` напрямую, вы все равно могли их вызвать, потому что вы сохранили возвращаемые значения при каждом вызове функции `trickOrTreat()` и вызвали функции с помощью переменных `trickFunction` и `treatFunction`.

```
Have a treat!  
No treats!
```


Теперь вы знаете, как функции могут возвращать другие функции. Вы также можете передать функцию в качестве аргумента другой функции. Возможно, вы хотите предоставить функции `trickOrTreat()` какое-то пользовательское поведение, чтобы она делала что-то еще, кроме возврата любой из двух строк. Функция, принимающая в качестве аргумента другую функцию, позволяет передавать разные функции при каждом ее вызове.

Передайте функцию другой функции в качестве аргумента.

В некоторых частях света, где празднуют Хэллоуин, дети получают мелочь вместо конфет или получают и то, и другое. Вы измените свою функцию `trickOrTreat()`, чтобы позволить передавать в качестве аргумента дополнительное угощение, представленное функцией.

Функция, которую `trickOrTreat()` использует в качестве параметра, также должна принимать собственный параметр. При объявлении типов функций параметры не обозначаются. Достаточно указать типы данных каждого параметра, разделив их запятой. Синтаксис показан на этом рисунке:

parameters  
(types only)



(String, Int) -> Int



Когда вы пишете лямбда-выражение для функции, которая принимает параметр, параметрам присваиваются имена в порядке их появления. Имена параметров перечисляются после открывающих фигурных скобок, и каждое имя разделяется запятой. Стрелка (->) отделяет имена параметров от тела функции. Синтаксис показан на этом рисунке:

```
val function name = { parameter 1 , parameter 2 ->
    function body
}
```

Обновите функцию `trickOrTreat()`, чтобы она принимала функцию в качестве параметра:

- После параметра `isTrick` добавьте дополнительный параметр `ExtraTreat` типа `(Int) -> String`.

```
``kt fun trickOrTreat(isTrick: Boolean, extraTreat: ((Int) -> String)?): () -> Unit {
```

В блоке `else`, перед оператором `return`, вызовите `println()`, передав в него вызов функции `extraTreat()`. В вызове функции `extraTreat()` передайте `5`.

```
``kt

fun trickOrTreat(isTrick: Boolean, extraTreat: (Int) -> String): () -> Unit {
    if (isTrick) {
        возвращает трюк
    } else {
        println(extraTreat(5))
        return treat
    }
}
```

Теперь при вызове функции `trickOrTreat()` вам нужно определить функцию с лямбда-выражением и передать ее для параметра `extraTreat`. В функции `main()` перед вызовом функции `trickOrTreat()` добавьте функцию `coins()`. Функция `coins()` присваивает параметру `Int` имя `quantity` и возвращает `String`. Вы можете заметить отсутствие ключевого слова `return`, которое не может быть использовано в лямбда-выражениях. Вместо этого возвращаемым значением становится результат последнего выражения в функции.

```
``kt fun main() { val coins: (Int) -> String = { quantity -> «$quantity quarters» }
```

```
val treatFunction = trickOrTreat(false)
val trickFunction = trickOrTreat(true)
```

```
treatFunction()
trickFunction()
```

```
}
```

После функции `coins()` добавьте функцию `cupcake()`, как показано на рисунке. Назовите параметр `Int` количеством и отделите его от тела функции с помощью оператора `->`. Теперь в функцию `trickOrTreat()` можно передать либо функцию `coins()`, либо функцию `cupcake()`.

```
```kt
fun main() {
    val coins: (Int) -> String = { quantity ->
        "$quantity quarters"
    }

    val cupcake: (Int) -> String = { quantity ->
        "Have a cupcake!"
    }

    val treatFunction = trickOrTreat(false)
    val trickFunction = trickOrTreat(true)
    treatFunction()
    trickFunction()
}
```

В функции `cupcake()` удалите параметр количества и символ `->`. Они не используются, поэтому их можно опустить.

```
```kt val cupcake: (Int) -> String = { «Съешьте кекс!» }
```

> Примечание: В функции `coins()` параметр `Int` назван количеством. Однако его можно назвать как угодно, лишь бы имя параметра и имя переменной в строке совпадали.

Обновите вызовы функции `trickOrTreat()`. Для первого вызова, когда `isTrick` равно `false`, передайте функцию `coins()`. Для второго вызова, когда `isTrick` равно `true`, передайте функцию `cupcake()`.

```
```kt
fun main() {
    val coins: (Int) -> String = { quantity ->
        "$quantity quarters"
    }

    val cupcake: (Int) -> String = {
        "Have a cupcake!"
    }
```

```

    }

    val treatFunction = trickOrTreat(false, coins)
    val trickFunction = trickOrTreat(true, cupcake)
    treatFunction()
    trickFunction()
}

```

Запустите свой код. Функция `extraTreat()` вызывается только в том случае, если параметр `isTrick` имеет значение `false`, поэтому на выходе получаем 5 четвертаков, но не кексы.

```
``kt 5 четвертаков Угощайтесь! Нет угощений
```

#### Нулевые типы функций

Как и другие типы данных, типы функций могут быть объявлены как `nullable`. В этом случае переменная может содержать функцию или быть нулевой.

Чтобы объявить функцию как `nullable`, окружите тип функции круглыми скобками, за которыми следует символ `?` вне завершающей скобки. Например, если вы хотите сделать тип `() -> String` обнуляемым, объявите его как тип `((() -> String)?`. Синтаксис `extraTreat` позволяет сделать параметр `nullable`, чтобы не создавать дополнительную функцию `Treat()` каждый раз, когда вы вызываете функцию `trickOrTreat()`: как показано на этом рисунке:



- Измените тип параметра `extraTreat` на `((() -> String)?`

```
``kt
fun trickOrTreat(isTrick: Boolean, extraTreat: (() -> String)?): () -> Unit {

```

- Измените вызов функции `extraTreat()`, чтобы использовать оператор `if` для вызова функции только в том случае, если она не является нулевой. Теперь функция `trickOrTreat()` должна выглядеть так, как показано в этом фрагменте кода:

```

fun trickOrTreat(isTrick: Boolean, extraTreat: (() -> String)?): () -> Unit {
    if (isTrick) {
        return trick
    } else {
        if (extraTreat != null) {
            println(extraTreat(5))
        }
        return treat
    }
}

```

- Удалите функцию `surprise()`, а затем замените аргумент `surprise` на `null` во втором вызове функции `trickOrTreat()`.

```
fun main() {  
    val coins: (Int) -> String = { quantity ->  
        "$quantity quarters"  
    }  
  
    val treatFunction = trickOrTreat(false, coins)  
    val trickFunction = trickOrTreat(true, null)  
    treatFunction()  
    trickFunction()  
}
```

- Запустите свой код. Результат должен остаться неизменным. Теперь, когда вы можете объявлять типы функций как nullable, вам больше не нужно передавать функцию для параметра `extraTreat`.

```
5 quarters  
Have a treat!  
No treats!
```

## Пишите лямбда-выражения с помощью сокращенного синтаксиса

Лямбда-выражения предоставляют множество способов сделать ваш код более лаконичным. В этом разделе вы изучите несколько из них, потому что большинство лямбда-выражений, с которыми вы сталкиваетесь и которые пишете, написаны с использованием сокращенного синтаксиса.

### Опустите имя параметра

Когда вы писали функцию `coins()`, вы явно объявили имя `quantity` для параметра `Int` функции. Однако, как вы видели на примере функции `surprise()`, имя параметра можно полностью опустить.

Когда у функции один параметр и вы не указываете его имя, Kotlin неявно присваивает ему имя `it`, поэтому вы можете опустить имя параметра и символ `->`, что делает ваши лямбда-выражения более лаконичными.

Синтаксис показан на этом изображении:

```
val coins: (Int) -> String = { quantity ->
    "$quantity quarters"
}
```



```
val coins: (Int) -> String = {
    "$it quarters"
}
```

Обновите функцию coins(), чтобы использовать сокращенный синтаксис для параметров:

- В функции coins() удалите имя параметра количества и символ ->.

```
val coins: (Int) -> String = {
    "$quantity quarters"
}
```

- Измените шаблон строки «\$quantity quarters», чтобы ссылаться на одиночный параметр с помощью \$it.

```
val coins: (Int) -> String = {
    "$it quarters"
}
```

- Запустите свой код. Kotlin распознает имя параметра it в качестве параметра Int и все равно выводит количество четвертей.

```
5 quarters
Have a treat!
No treats!
```

Передайте лямбда-выражение непосредственно в функцию

Функция coins() в настоящее время используется только в одном месте. Что, если бы вы могли просто передать лямбда-выражение непосредственно в функцию trickOrTreat() без необходимости

предварительно создавать переменную?

Лямбда-выражения - это просто литералы функций, точно так же, как 0 - целочисленный литерал или «Hello» - строковый литерал. Вы можете передать лямбда-выражение непосредственно в вызов функции. Синтаксис показан на этом рисунке:

```
var coins = Lambda expression  
trickOrTreat(false, coins)  
↓  
trickOrTreat(false, Lambda expression)
```

Измените код так, чтобы удалить переменную coins:

- Переместите лямбда-выражение так, чтобы оно передавалось непосредственно в вызов функции trickOrTreat(). Вы также можете сократить лямбда-выражение до одной строки.

```
fun main() {  
    val coins: (Int) -> String = {  
        "$it quarters"  
    }  
    val treatFunction = trickOrTreat(false, { "$it quarters" })  
    val trickFunction = trickOrTreat(true, null)  
    treatFunction()  
    trickFunction()  
}
```

- Удалите переменную coins, потому что она больше не используется.

```
fun main() {  
    val treatFunction = trickOrTreat(false, { "$it quarters" })  
    val trickFunction = trickOrTreat(true, null)  
    treatFunction()  
    trickFunction()  
}
```

- Запустите код. Он по-прежнему компилируется и выполняется, как и ожидалось.

```
5 quarters  
Have a treat!
```

```
No treats!
```

## Использовать синтаксис лямбды в конце

Вы можете использовать другой вариант написания лямбд, когда тип функции является последним параметром функции. В этом случае вы можете поместить лямбда-выражение после закрывающей круглой скобки, чтобы вызвать функцию. Синтаксис показан на этом рисунке:

The diagram illustrates a syntactic transformation for lambda expressions in Kotlin. It shows two equivalent ways to write a function call. The top line shows `trickOrTreat(false, { })`, where the lambda expression `{ }` is inside the parentheses. An arrow points down to the bottom line, which shows `trickOrTreat(false) { }`, where the lambda expression is placed after the closing parenthesis. In both cases, the lambda expression is highlighted with a blue rounded rectangle and labeled "Lambda expression".

Это делает ваш код более читабельным, поскольку отделяет лямбда-выражение от других параметров, но не меняет того, что делает код.

Обновите код, чтобы использовать синтаксис лямбды с конца:

- В переменной `treatFunction` переместите лямбда-выражение `{ "$it quarters" }` после закрывающей скобки в вызове функции `trickOrTreat()`.

```
val treatFunction = trickOrTreat(false) { "$it quarters" }
```

- Запустите свой код. Все по-прежнему работает!

```
5 quarters  
Have a treat!  
No treats!
```

Примечание: Композитные функции, которые вы использовали для объявления вашего пользовательского интерфейса, принимают функции в качестве параметров и обычно вызываются с помощью синтаксиса лямбды.

## Используйте функцию `repeat()`

Когда функция возвращает функцию или принимает функцию в качестве аргумента, она называется функцией высшего порядка.

Функция `trickOrTreat()` является примером функции высшего порядка, потому что она принимает в качестве параметра функцию типа `((Int) -> String)?` и возвращает функцию типа `() -> Unit`. Kotlin

предоставляет несколько полезных функций высшего порядка, которыми вы можете воспользоваться, используя свои новые знания о лямбдах.

Функция `repeat()` - одна из таких функций высшего порядка. Функция `repeat()` - это лаконичный способ выразить цикл `for` с помощью функций. В последующих разделах вы будете часто использовать эту и другие функции высшего порядка. Функция `repeat()` имеет такую сигнатуру:

```
repeat(times: Int, action: (Int) -> Unit)
```

Параметр `times` - это количество раз, которое должно произойти действие. Параметр `action` - это функция, которая принимает один параметр `Int` и возвращает тип `Unit`. Параметр `Int` функции `action` - это количество раз, которое действие выполнилось на данный момент, например аргумент `0` для первой итерации или аргумент `1` для второй итерации. Функцию `repeat()` можно использовать для повторения кода заданное количество раз, подобно циклу `for`. Синтаксис показан на этом рисунке:

The diagram illustrates the transformation of a `for` loop into a `repeat` function call. At the top, a `for` loop is shown with its components highlighted in blue boxes: `iteration` (the loop variable), `start` (the beginning of the range), and `end` (the end of the range). The loop body contains `// code`. A downward arrow indicates the transformation. Below, the equivalent `repeat` function call is shown, with `times` (the number of iterations) and `iteration` (the lambda parameter) highlighted in blue boxes. The lambda body contains `// code`.

```
for ( iteration in start . . end ) {  
    // code  
}
```

↓

```
repeat ( times ) { iteration ->  
    // code  
}
```

Вместо того чтобы вызывать функцию `trickFunction()` только один раз, вы можете вызвать ее несколько раз с помощью функции `repeat()`.

Обновите код фокуса, чтобы увидеть функцию `repeat()` в действии:

- В функции `main()` вызовите функцию `repeat()` в промежутке между вызовами функций `treatFunction()` и `trickFunction()`. Для параметра `times` передайте значение `4`, а для функции действия используйте синтаксис лямбды в конце. Не нужно указывать имя параметра `Int` лямбда-выражения.



```
fun main() {  
    val treatFunction = trickOrTreat(false) { "$it quarters" }  
    val trickFunction = trickOrTreat(true, null)  
    treatFunction()  
    trickFunction()  
    repeat(4) {  
  
    }  
}
```

- Переместите вызов функции `treatFunction()` в лямбда-выражение функции `repeat()`.

```
fun main() {  
    val treatFunction = trickOrTreat(false) { "$it quarters" }  
    val trickFunction = trickOrTreat(true, null)  
    repeat(4) {  
        treatFunction()  
    }  
    trickFunction()  
}
```

- Запустите свой код. Строка «Have a treat» должна напечататься четыре раза.

```
5 quarters  
Have a treat!  
Have a treat!  
Have a treat!  
Have a treat!  
No treats!
```

## Заключение

Вы познакомились с основами типов функций и лямбда-выражений. Знакомство с этими понятиями поможет вам в дальнейшем изучении языка Kotlin. Использование типов функций, функций высшего порядка и сокращенного синтаксиса также делает ваш код более кратким и легким для чтения.

## Резюме

- Функции в Kotlin являются первоклассными конструкциями и могут рассматриваться как типы данных.
- Лямбда-выражения предоставляют сокращенный синтаксис для написания функций.
- Вы можете передавать типы функций в другие функции.
- Вы можете возвращать тип функции из другой функции.
- Лямбда-выражение возвращает значение последнего выражения.
- Если в лямбда-выражении с одним параметром метка параметра опущена, на него ссылаются с помощью идентификатора `it`.

- Лямбды можно записывать в строку без имени переменной.
- Если последний параметр функции является типом функции, можно использовать синтаксис `trailing lambda` для перемещения лямбда-выражения после последней круглой скобки при вызове функции.
- Функции высшего порядка - это функции, которые принимают в качестве параметров другие - функции или возвращают функцию.
- Функция `repeat()` - это функция высшего порядка, которая работает аналогично циклу `for`.