## **Высокоуровневые функции и лямбды**

**Функция высшего порядка** - это функция, которая принимает функции как параметры, или возвращает функцию в качестве результата.

Хорошим примером такой функции является lock(), которая берёт блокирующий объект и функцию, получает блокировку, выполняет функцию и отпускает блокировку:

**fun** **<T>** lock(lock: **Lock, body: (**) -> T): T{

lock.lock()

**try**{

**return** body()

}

**finally** {

lock.unlock()

}

}

Проанализируем этот код: body имеет функциональный тип: () -> T, то есть параметр должен быть функцией без параметров, возвращающей значение типа T. Она вызывается внутри блока try, под защитой объекта lock, получившего блокировку вызовом функции lock().

Если мы хотим вызвать метод lock(), можно передать другую функцию в качестве входящего аргумента:

**fun** toBeSynchronized() = sharedResource.operation()

**val** result = lock (lock, ::toBeSynchronized)

Обычно удобней передавать [лямбда-выражения](https://kotlinlang.ru/docs/reference/lambdas.html#lambda-expressions-and-anonymous-functions):

**val** result = lock(lock, { sharedResource.operation() })

Лямбда-выражения подробно описаны [здесь](https://kotlinlang.ru/docs/reference/lambdas.html#lambda-expressions-and-anonymous-functions), сделаем краткий обзор:

* Лямбда-выражение всегда заключено в фигурные скобки;
* Его параметры (если они есть) объявлены до знака -> (допустимо не указывать);
* Тело выражения следует после знака ->.

В Kotlin есть конвенция, согласно которой, если последний параметр функции является функцией, которая передается в виде лямбда-выражения, можно вынести его за скобки:

lock (lock) { sharedResource.operation() }

*И так это одно и тоже:*

**fun** toBeSynchronized() = sharedResource.operation()

**val** result = lock (lock, ::toBeSynchronized)

**val** result = lock(lock, { sharedResource.operation() })

**val** result = lock (lock) { sharedResource.operation() }

Следующим примером функции высшего порядка выберем функцию map():

**fun** **<T, R>** List**<T>**.map(transform: **(T**) -> R): List**<R>** {

**val** result = arrayListOf<R>()

**for** (item **in** this)

result.add(transform(item))

**return** result

}

Эту функцию можно вызвать так:

**val** doubled = ints.map { it -> it \* 2 }

Обратите внимание, что скобки можно вообще не указывать при вызове функции, если лямбда является единственным аргументом.

**val** doubled = ints.map { it \* 2 }

## **Ключевое слово it: неявное имя единственного параметра**

Ещё одна полезная конвенция состоит в том, что если функциональный литерал имеет ровно один параметр, его объявление можно удалить (вместе с ->), и обращаться к нему по имени it:

ints.map { it \* 2 }

Эти конвенции позволяют писать код в стиле [LINQ](http://msdn.microsoft.com/en-us/library/bb308959.aspx) (**интегрированный запрос на языке .NET)**:

strings.filter { it.length == 5 }.sortBy { it }.map { it.toUpperCase() }

## **Символ подчеркивания для неиспользуемых переменных**

Если параметр лямбды не используется, разрешено применять подчеркивание вместо его имени

map.forEach { \_, value -> println(**"$value!"**) }

## **Деструктуризация в лямбдах**

Деструктуризация в лямбдах описана в [деструктурирующие объявления](https://kotlinlang.ru/docs/reference/multi-declarations.html#destructuring-in-lambdas-since-11).

Этот синтаксис называется деструктуризирующее присваивание. Он позволяет присвоить объект сразу нескольким переменным, разбив его на части. Мы объявили две переменные: name и age, и теперь можем использовать их по отдельности:

**val** (name, age) = person

println(name)

println(age)

Эта декларация транслируется в такой код:

**val** name = person.component1()

**val** age = person.component2()

Далее СМ ВНИЗУ

## **Инлайн - Встроенные (inline) функции**

Иногда выгодно улучшить производительность функций высшего порядка, используя [инлайн функции](https://kotlinlang.ru/docs/reference/inline-functions.html).

Например, функция lock() может быть встроена в то место, из которого она вызывается:

lock(l) { foo() }

Вместо создания объекта функции для параметра и генерации вызова, компилятор может выполнить:

l.lock()

**try** {

foo()

}

**finally** {

l.unlock()

}

Чтобы заставить компилятор поступить именно так, нам необходимо отметить функцию lock модификатором inline, Т.е. помечается ФУНКЦИЯ как встраиваемая:

**inline** **fun** **<T>** lock(lock: **Lock, body: (**) -> T): T {

*// ...*

}

Далее СМ ВНИЗУ

## **Лямбда-выражения и анонимные функции**

Лямбда-выражение, или анонимная функция, это "функциональный литерал", то есть необъявленная функция, которая немедленно используется в качестве выражения. Рассмотрим следующий пример:

max(strings, { a, b -> a.length < b.length })

Функция max является функцией высшего порядка, потому что она принимает функцию в качестве второго аргумента. Этот второй аргумент является выражением, которое в свою очередь есть функция, то есть *функциональный литерал*. Как функция он эквивалентен объявлению:

**fun** compare(a: **String, b: String**): **Boolean** = a.length **< b.length**

## **Функциональные типы**

Чтобы функция могла принять функцию в качестве параметра, необходимо указать тип функции-параметра. Например, вышеуказанная функция max определена так:

**fun** **<T>** max(collection: **Collection<T>, less: (T, T**) -> **Boolean**): T? {

**var** max: T? = **null**

**for** (it **in** collection)

**if** (max == **null** || less(max, it))

max = it

**return** max

}

Параметр less является (T, T) -> Boolean типом, то есть функцией, которая принимает два параметра типа T и возвращает Boolean: 'true', если первый параметр меньше второго.

В теле функции, строка 4, less используется как функция: она вызвана с двумя параметрами типа T.

Функциональный тип записывается, как указано выше, или может иметь именованные параметры, если нужно выявить смысл каждого из параметров.

**val** compare: (x: T, y: T) -> **Int** = ...

## **Синтаксис лямбда-выражений**

Полная синтаксическая форма лямбда-выражений, таких как *literals of function types*, может быть представлена следующим образом:

**val** sum = { x: **Int**, y: **Int** -> x + y }

Лямбда-выражение всегда заключено в скобки {...}, объявление параметров при таком синтаксисе происходит внутри этих скобок и может включать в себя аннотации типов (опционально), тело функции начинается после знака ->. Если тип возвращаемого значения не Unit, то в качестве возвращаемого типа принимается последнее (а возможно и единственное) выражение внутри тела лямбды.

Если мы вынесем все необязательные объявления, то, что останется, будет выглядеть следующим образом:

**val** sum: (**Int**, **Int**) -> **Int** = { x, y -> x + y }

Обычное дело, когда лямбда-выражение имеет только один параметр. Если **Kotlin** может определить сигнатуру метода сам, он позволит нам не объявлять этот единственный параметр, и объявит его сам под именем it:

ints.filter { it > 0 } *//Эта константа имеет тип '(it: Int) -> Boolean'*

Мы можем явно вернуть значение из лямбды, используя [qualified return](https://kotlinlang.ru/docs/reference/returns.html" \l "return-at-labels) синтаксис:

ints.filter {

**val** shouldFilter = it > 0

shouldFilter

}

ints.filter {

**val** shouldFilter = it > 0

**return**@filter shouldFilter

}

Обратите внимание, что функция принимает другую функцию в качестве своего последнего параметра, аргумент лямбда-выражения в таком случае может быть принят вне списка аргументов, заключённого в скобках. См. [callSuffix](https://kotlinlang.ru/docs/reference/grammar.html" \l "call-suffix).

## **Анонимные функции**

Единственной особенностью синтаксиса лямбда-выражений, о которой ещё не было сказано, является способность определять и назначать возвращаемый функцией тип. В большинстве случаев в этом нет особой необходимости, потому что он может быть вычислен автоматически. Однако, если у вас есть потребность в определении возвращаемого типа, вы можете воспользоваться альтернативным синтаксисом:

fun(x: **Int, y: Int**): **Int** = x + y

Объявление анонимной функции выглядит очень похоже на обычное объявление функции, за исключением того, что её имя опущено. Тело такой функции может быть описано и выражением (как показано выше), и блоком:

fun(x: **Int, y: Int**): **Int** {

**return** x + y

}

Параметры функции и возвращаемый тип обозначаются таким же образом, как в обычных функциях. Правда, тип параметра может быть опущен, если его значение следует из контекста:

ints.filter(fun(item) = item > 0)

Аналогично и с типом возвращаемого значения: он вычисляется автоматически для функций-выражений или же должен быть определён вручную (если не является типом Unit) для анонимных функций, которые имеют в себе блок.

Обратите внимание, что параметры анонимных функций всегда заключены в круглые скобки (...). Приём, позволяющий оставлять параметры вне скобок, работает только с лямбда-выражениями.

Одним из отличий лямбда-выражений от анонимных функций является поведение оператора return ([non-local returns](https://kotlinlang.ru/docs/reference/inline-functions.html" \l "non-local-returns)). Слово return , не имеющее метки (@), всегда возвращается из функции, объявленной ключевым словом fun. Это означает, что return внутри лямбда-выражения возвратит выполнение к функции, включающей в себя это лямбда-выражение. Внутри анонимных функций оператор return, в свою очередь, выйдет, собственно, из анонимной функции.

## **Замыкания**

Лямбда-выражение или анонимная функция (так же, как и [локальная функция](https://kotlinlang.ru/docs/reference/functions.html#local-functions) или [object expression](https://kotlinlang.ru/docs/reference/object-declarations.html" \l "object-expressions)) имеет доступ к своему замыканию, то есть к переменным, объявленным вне этого выражения или функции. В отличие от Java, переменные, захваченные в замыкании, могут быть изменены:

**var** sum = 0

ints.filter { it > 0 }.forEach {

sum += it

}

print(sum)

## **Литералы функций с объектом-приёмником**

Kotlin предоставляет возможность вызывать литерал функции с указанным объектом-приёмником. Внутри тела литерала вы можете вызывать методы объекта-приёмника без дополнительных определителей. Это схоже с принципом работы [расширений](https://kotlinlang.ru/docs/reference/extensions.html), которые позволяют получить доступ к членам объекта-приёмника внутри тела функции. Один из самых важных примеров использования литералов с объектом-приёмником это [Type-safe Groovy-style builders](https://kotlinlang.ru/docs/reference/type-safe-builders.html).

Тип такого литерала — это тип функции с приёмником:

sum : **Int**.(other: **Int**) -> **Int**

По аналогии с расширениями, литерал функции может быть вызван так, будто он является методом объекта-приёмника:

1.sum(2)

Синтаксис анонимной функции позволяет вам явно указать тип приёмника. Это может быть полезно в случае, если вам нужно объявить переменную типа нашей функции для использования в дальнейшем.

**val** sum = **fun** **Int**.(other: **Int**): **Int** = this + other

Лямбда-выражения могут быть использованы как литералы функций с приёмником, когда тип приёмника может быть выведен из контекста.

**class** HTML {

**fun** body() { ... }

}

**fun** html(init: **HTML.(**) -> **Unit**): HTML {

**val** html = HTML() *// создание объекта-приёмника*

html.init() *// передача приёмника в лямбду*

**return** html

}

html { *// лямбда с приёмником начинается тут*

body() *// вызов метода объекта-приёмника*

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## **Деструктуризация в лямбдах**

Деструктуризация в лямбдах описана в [деструктурирующие объявления](https://kotlinlang.ru/docs/reference/multi-declarations.html#destructuring-in-lambdas-since-11).

Этот синтаксис называется деструктуризирующее присваивание. Он позволяет присвоить объект сразу нескольким переменным, разбив его на части. Мы объявили две переменные: name и age, и теперь можем использовать их по отдельности:

**val** (name, age) = person

println(name)

println(age)

Эта декларация транслируется в такой код:

**val** name = person.component1()

**val** age = person.component2()

Как и многое другое в Kotlin, мульти-декларации опираются на конвенцию: функции componentN() вызываются по имени, то есть могут быть объявлены как в классе person, так и вне его — в качестве [расширений](https://kotlinlang.ru/docs/reference/extensions.html).

Заметьте, что функции componentN() нужно отмечать ключевым словом operator, чтобы позволить их использование в деструктуризирующем присваивании.

Деструктуризирующие присваивания также работают в циклах for:

**for** ((a, b) **in** collection) { ... }

В данном примере значения переменных a и b возращены методами component1() и component2(), вызванными неявно у элементов коллекции.

## **Например: возврат двух значений из функции**

Предположим, нам нужно вернуть два значения из функции. Например, результат вычисления и какой-нибудь статус. Компактный способ достичь этого — объявление data-класса и возвращение его экземпляра:

data **class** Result(**val** result: **Int**, **val** status: Status)

**fun** function(...): Result {

*// вычисления*

**return** Result(result, status)

}

*// Теперь мы можем использовать деструктуризирующее присваивание:*

**val** (result, status) = function(...)

Так как data-классы автоматически объявляют componentN()-функции, мульти-декларации будут работать с ними "из коробки".

**ПРИМЕЧАНИЕ**: мы также могли использовать стандартный класс Pair, чтобы заставить функцию вернуть Pair<Int, Status>, но правильнее будет именовать ваши данные должным образом.

Пожалуй, самый хороший способ итерации по ассоциативному списку:

**for** ((key, value) **in** map) {

*// do something with the key and the value*

}

Чтобы это работало, мы должны:

* представить ассоциативный список как последовательность значений, предоставив функцию iterator(),
* представить каждый элемент как пару с помощью функций component1() и component2().

И да, стандартная библиотека предоставляет такие расширения:

operator **fun** **<K, V>** Map**<K, V>**.iterator(): Iterator**<Map.Entry<K, V>**>=entrySet().iterator()

operator **fun** **<K, V>** Map.Entry**<K, V>**.component1() = getKey()

operator **fun** **<K, V>** Map.Entry**<K, V>**.component2() = getValue()

Так что вы можете свободно использовать мульти-декларации в циклах for с ассоциативными списками (так же как и с коллекциями экземпляров data-классов).

## **Встроенные (inline) функции**

Использование [функций высшего порядка](https://kotlinlang.ru/docs/reference/lambdas.html) влечёт за собой снижение производительности: во-первых, функция является объектом, а во-вторых, происходит захват контекста замыканием, то есть функции становятся доступны переменные, объявленные вне её тела. А выделения памяти (как для объекта функции, так и для её класса) и виртуальные вызовы занимают системные ресурсы.

Но во многих случаях эти "накладные расходы" можно устранить с помощью инлайнинга (встраивания) лямбда-выражений. Например, функция lock() может быть легко встроена в то место, из которого она вызывается:

lock(l) { foo() }

Вместо создания объекта функции для параметра и генерации вызова, компилятор мог бы выполнить что-то подобное этому коду:

l.lock()

**try** {

foo()

}

**finally** {

l.unlock()

}

Разве это не то, чего мы хотели изначально?

Чтобы заставить компилятор поступить именно так, нам необходимо отметить функцию lock модификатором inline:

**inline** **fun** **<T>** lock(lock: **Lock, body: (**) -> T): T {

*// ...*

}

Модификатор inline влияет и на функцию, и на лямбду, переданную ей: они обе будут встроены в место вызова.

Встраивание функций может увеличить количество сгенерированного кода, но если вы будете делать это в разумных пределах (не инлайнить большие функции), то получите прирост производительности, особенно при вызове функций с параметрами разного типа внутри циклов.

**noinline**

В случае, если вы хотите, чтобы только некоторые лямбды, переданные inline-функции, были встроены, вам необходимо отметить модификатором noinline те функции-параметры, которые встроены не будут:

**inline** **fun** foo(inlined: **(**) -> **Unit**, noinline notInlined: () -> **Unit**) {

*// ...*

}

Когда как встраиваемые лямбды могут быть вызваны только внутри inline-функций или переданы в качестве встраиваемых аргументов, с noinline-функциями можно работать без ограничений: хранить внутри полей, передавать куда-либо и т.д.

Заметьте, что если inline-функция не имеет ни inline параметров, ни [параметров вещественного типа](https://kotlinlang.ru/docs/reference/inline-functions.html#%D0%BF%D0%B0%D1%80%D0%B0%D0%BC%D0%B5%D1%82%D1%80%D1%8B-%D0%B2%D0%B5%D1%89%D0%B5%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%BE%D0%B3%D0%BE-%D1%82%D0%B8%D0%BF%D0%B0), компилятор выдаст предупреждение, так как встраивание такой функции вряд ли принесёт пользу (вы можете скрыть предупреждение, если уверены, что встраивание необходимо).

**Нелокальные return**

В Kotlin мы можем использовать обыкновенный, безусловный return только для выхода из именованной функции или анонимной функции. Это значит, что для выхода из лямбды нам нужно использовать [label](https://kotlinlang.ru/docs/reference/returns.html" \l "return-at-labels). Обычный return запрещён внутри лямбды, потому что она не может заставить внешнюю функцию завершиться.

**fun** foo() {

ordinaryFunction {

**return** *// ERROR: can not make `foo` return here*

}

}

Но если функция, в которую передана лямбда, встроена, то return также будет встроен, поэтому так делать можно:

**fun** foo() {

inlineFunction {

**return** *// OK: the lambda is inlined*

}

}

Такие return (находящиеся внутри лямбд, но завершающие внешнюю функцию) называются нелокальными (non-local). Мы используем такие конструкции в циклах, которые являются inline-функциями:

**fun** hasZeros(ints: **List<Int>**): **Boolean** {

ints.forEach {

**if** (it == 0) **return** **true** *// returns from hasZeros*

}

**return** **false**

}

Заметьте, что некоторые inline-функции могут вызывать переданные им лямбды не напрямую в теле функции, а из иного контекста, такого как локальный объект или вложенная функция. В таких случаях, нелокальное управление потоком выполнения также запрещено в лямбдах. Чтобы указать это, параметр лямбды необходимо отметить модификатором crossinline:

**inline** **fun** f(crossinline body: **(**) -> **Unit**) {

**val** f = **object**: Runnable {

**override** **fun** run() = body()

}

*// ...*

}

break и continue пока что недоступны во встроенных лямбдах, но мы планируем добавить их поддержку

**Параметры вещественного типа**

Иногда нам необходимо получить доступ к типу, переданному в качестве параметра:

**fun** **<T>** TreeNode.findParentOfType(clazz: **Class<T>**): T? {

**var** p = parent

**while** (p != **null** && !clazz.isInstance(p)) {

p = p?.parent

}

@Suppress(**"UNCHECKED\_CAST"**)

**return** p **as** T

}

В этом примере мы осуществляем проход по дереву и используем рефлексию, чтобы проверить узел на принадлежность к определённому типу. Это прекрасно работает, но вызов выглядит не очень симпатично:

myTree.findParentOfType(MyTreeNodeType::**class**.java)

Что мы на самом деле хотим, так это передать этой функции тип, то есть вызвать её вот так:

myTree.findParentOfType<MyTreeNodeType>()

В таких случаях inline-функции могут принимать *параметры вещественного типа* (reified type parameters). Чтобы включить эту возможность, мы можем написать что-то вроде этого:

**inline** **fun** **<reified T>** TreeNode.findParentOfType(): T? {

**var** p = parent

**while** (p != **null** && p !**is** T) {

p = p?.parent

}

**return** p **as** T

}

Мы определили тип параметра с помощью модификатора reified, но он доступен внутри функции почти так же, как и обычный класс. Так как функция встроена, то для работы таких операторов как !is и as рефлексия не нужна. Также, мы можем вызывать её таким же образом, как было упомянуто выше: myTree.findParentOfType<MyTreeNodeType>()

Хотя рефлексия может быть не нужна во многих случаях, мы всё ещё можем использовать её с параметром вещественного типа:

**inline** **fun** **<reified T>** membersOf() = T::**class**.members

**fun** main(s: **Array<String>**) {

println(membersOf<StringBuilder>().joinToString(**"\n"**))

}

Обычная функция (не отмеченная как встроенная) не может иметь параметры вещественного типа. Тип, который не имеет представление во времени исполнения (например, параметр невещественного или фиктивного типа вроде Nothing), не может использоваться в качестве аргумента для параметра вещественного типа.