**Программирование на языке Java**

**Лямбда выражения**

**Contents**

[1. λ – выражения 1](#_Toc498527236)

[1.1. Отложенное выполнение 7](#_Toc498527237)

[1.2. Передача параметров в лямбда-выражение 7](#_Toc498527238)

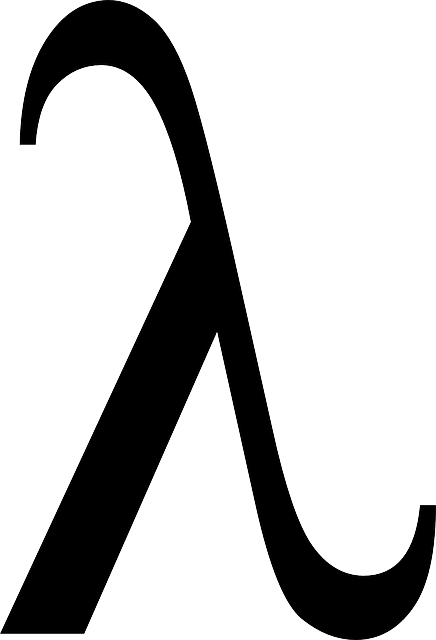
[1.3. Терминальные лямбда-выражения 8](#_Toc498527239)

[1.4. Лямбды и локальные переменные 8](#_Toc498527240)

[1.5. Блоки кода в лямбда-выражениях 9](#_Toc498527241)

[1.6. Обобщенный функциональный интерфейс 9](#_Toc498527242)

# λ – выражения



**λ**(**лямбда)-выражение** в программировании – специальный синтаксис для определения [функциональных объектов](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82), заимствованный из [λ-исчисления](https://ru.wikipedia.org/wiki/%CE%9B-%D0%B8%D1%81%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5). Применяется как правило для объявления [анонимных функций](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%BE%D0%BD%D0%B8%D0%BC%D0%BD%D0%B0%D1%8F_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F) по месту их использования, и обычно допускает [замыкание](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%BC%D1%8B%D0%BA%D0%B0%D0%BD%D0%B8%D0%B5_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29) на [лексический контекст](https://ru.wikipedia.org/wiki/%D0%9B%D0%B5%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%BA%D0%BE%D0%BD%D1%82%D0%B5%D0%BA%D1%81%D1%82), в котором это выражение использовано. Используя лямбда-выражения, можно объявлять функции в любом месте кода.

**Ля́мбда-исчисле́ние** (*λ-исчисление*) — [формальная система](https://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D1%80%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), разработанная американским математиком [Алонзо Чёрчем](https://ru.wikipedia.org/wiki/%D0%A7%D1%91%D1%80%D1%87,_%D0%90%D0%BB%D0%BE%D0%BD%D0%B7%D0%BE), для [формализации](https://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D1%80%D0%BC%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F) и анализа понятия [вычислимости](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D1%8C).

**Замыкание** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA" \o "Английский язык) *closure*) в программировании – [функция первого класса](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8_%D0%BF%D0%B5%D1%80%D0%B2%D0%BE%D0%B3%D0%BE_%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D0%B0), в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции в окружающем коде и не являющиеся её параметрами. Говоря другим языком, замыкание – функция, которая ссылается на [свободные переменные](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D0%BE%D0%B1%D0%BE%D0%B4%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F) в своём [контексте](https://ru.wikipedia.org/wiki/%D0%9B%D0%B5%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%BA%D0%BE%D0%BD%D1%82%D0%B5%D0%BA%D1%81%D1%82).

Замыкание, также как и [экземпляр объекта](https://ru.wikipedia.org/wiki/%D0%AD%D0%BA%D0%B7%D0%B5%D0%BC%D0%BF%D0%BB%D1%8F%D1%80_%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%B0), есть способ представления функциональности и данных, связанных и упакованных вместе.

Замыкание — это особый вид функции. Она определена в теле другой функции и создаётся каждый раз во время её выполнения. Синтаксически это выглядит как функция, находящаяся целиком в теле другой функции. При этом вложенная внутренняя функция содержит ссылки на локальные переменные внешней функции.

В [информатике](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0) [язык программирования](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) имеет **функции первого класса**, если он рассматривает функции как [объекты первого класса](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_%D0%BF%D0%B5%D1%80%D0%B2%D0%BE%D0%B3%D0%BE_%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D0%B0). В частности, это означает, что язык поддерживает передачу функций в качестве аргументов другим функциям, возврат их как результат других функций, присваивание их переменным или сохранение в структурах данных[[1]](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8_%D0%BF%D0%B5%D1%80%D0%B2%D0%BE%D0%B3%D0%BE_%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D0%B0#cite_note-1). Некоторые теоретики языков программирования считают необходимым условием также поддержку [анонимных функций](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%BE%D0%BD%D0%B8%D0%BC%D0%BD%D1%8B%D0%B5_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8)[[2]](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8_%D0%BF%D0%B5%D1%80%D0%B2%D0%BE%D0%B3%D0%BE_%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D0%B0#cite_note-test-2). В языках с функциями первого класса имена функций не имеют никакого специального статуса, они рассматриваются как обычные значения, тип которых является [функциональным](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D1%82%D0%B8%D0%BF)[[3]](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8_%D0%BF%D0%B5%D1%80%D0%B2%D0%BE%D0%B3%D0%BE_%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D0%B0#cite_note-3).

**Лексический контекст** (область видимости). В [программировании](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), **область видимости** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *scope*) обозначает область [программы](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0), в пределах которой [идентификатор](https://ru.wikipedia.org/wiki/%D0%98%D0%B4%D0%B5%D0%BD%D1%82%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%82%D0%BE%D1%80) (имя) некоторой [переменной](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29) продолжает быть связанным с этой переменной и возвращать её значение. За пределами области видимости тот же самый идентификатор может быть связан с другой переменной, либо быть свободным (не связанным ни с какой из них).

В большинстве [языков программирования](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) область видимости переменной определяется местом её объявления. Кроме того, область видимости может задаваться явно с помощью [классов памяти](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D0%B8) или [пространств имён](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D1%81%D1%82%D0%B2%D0%BE_%D0%B8%D0%BC%D1%91%D0%BD_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29).

**Анонимная функция** в [программировании](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) — особый вид [функций](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29), которые объявляются в месте использования и не получают уникального [идентификатора](https://ru.wikipedia.org/wiki/%D0%98%D0%B4%D0%B5%D0%BD%D1%82%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%82%D0%BE%D1%80) для доступа к ним. Поддерживаются во многих [языках программирования](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F).

Обычно при создании анонимные функции либо вызываются напрямую, либо [ссылка](https://ru.wikipedia.org/wiki/%D0%A1%D1%81%D1%8B%D0%BB%D0%BA%D0%B0_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29) на функцию присваивается [переменной](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29), с помощью которой затем можно косвенно вызывать данную функцию. Если анонимная функция ссылается на переменные, не содержащиеся в её теле (захват), то такая функция называется [замыканием](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%BC%D1%8B%D0%BA%D0%B0%D0%BD%D0%B8%D0%B5_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29). [Лямбда-выражение](https://ru.wikipedia.org/wiki/%D0%9B%D1%8F%D0%BC%D0%B1%D0%B4%D0%B0-%D0%B2%D1%8B%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5) — типичная для многих языков [синтаксическая](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%81_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29) конструкция для определения анонимной функции.

В программировании анонимная функция (**функциональный литерал, лямбда выражение**) является функцией, определение которой не привязано ни к какому идентификатору. В анонимных функциях часто:[[1]](https://en.wikipedia.org/wiki/Anonymous_function#cite_note-1)

1. аргументы передаются функциям высшего порядка , или
2. используются в качестве возвращаемого значения функции высшего порядка.

λ(лямбда)-выражение представляет набор инструкций, которые можно выделить в отдельную переменную и затем многократно вызвать в различных местах программы.

Основу лямбда-выражения составляет лямбда-оператор, который представляет стрелку ->. Этот оператор разделяет лямбда-выражение на две части: левая часть содержит список формальных параметров выражения заключенных в круглые скобки, а правая собственно представляет тело лямбда-выражения, где выполняются все действия. В лямбда выражении можно опускать типы параметров, можно опустить и скобки, если параметр один. Тело лямбда выражения может состоять из нескольких операторов, которые образуют блок (операторы, заключенные в фигурные скобки).

A return statement is not an expression; in a lambda expression, you must enclose statements in braces ({}). However, you do not have to enclose a void method invocation in braces. For example, the following is a valid lambda expression:

Примеры лямбда выражений (анонимных функций) в Java.

// with no parameter

() -> System.out.println("Hello, world.");

// with a single parameter (This example is an identity function).

a -> a

email -> System.out.println(email)

//with a single expression

(a, b) -> a + b

// with explicit type information

(Long id, String name) -> "id: " + id + ", name:" + name

// with a code block

(a, b) -> {**return** a + b;}

// with multiple statements in the lambda body. It require a code block.

// This example also includes a nested lambda expression as well as a closure.

(id, newPrice) -> {

Optional<Product> mayBeProduct = findProduct(id);

mayBeProduct.ifPresent(product -> product.setPrice(newPrice));

**return** mayBeProduct.get();

}

p -> {

return p.getGender() == Person.Sex.MALE

&& p.getAge() >= 18

&& p.getAge() <= 25;

}

// with multiple statements in the lambda body. It needs a code block.

// This example also includes two nested lambda expressions (the first one is also a closure).

(id, defaultPrice) -> {

Optional<Product> product = productList.stream().filter(p -> p.getId() == id).findFirst();

**return** product.map(p -> p.getPrice()).orElse(defaultPrice);

}

Лямбда-выражение не выполняется само по себе, а образует реализацию метода, определенного в функциональном интерфейсе. При этом важно, что функциональный интерфейс должен содержать только один метод без реализации.

**public** **class** Lambda {

**public** **static** **void** main(String[] args) {

BinOp operation;

operation = (x,y)->x+y;

**int** result = operation.calculate(10, 20);

System.***out***.println(result); //30

}

**interface** BinOp{

**int** calculate(**int** x, **int** y);

}

}

В роли функционального интерфейса выступает интерфейс BinOP, в котором определен один метод без реализации - метод calculate(). Данный метод принимает два параметра - целых числа, и возвращает некоторое целое число.

Использование лямбда-выражений похоже на использование внутренних анонимных классов. В частности, предыдущий пример можно переписать с использованием внутреннего анонимного класса::

**public** **class** Lambda {

**public** **static** **void** main(String[] args) {

BinOp op = **new** BinOp(){

**public** **int** calculate(**int** x, **int** y){

**return** x + y;

}

};

**int** z = op.calculate(20, 10);

System.***out***.println(z); // 30

}

}

**interface** BinOp{

**int** calculate(**int** x, **int** y);

}

Лямбда выражения конвертируются в “функциональные интерфейсы” (определяемые как интерфейсы, которые содержат только один абстрактный метод в дополнение к одному или нескольким методам по умолчанию (default) или статическим (static) методам), как это показано в следующем [примере:](https://en.wikipedia.org/wiki/Anonymous_function#Java)

**public** **class** Lambda {

**interface** IntegerMath {

**int** operation(**int** a, **int** b);

//Метод, реализованный в интерфейсе, называется методом по

//умолчанию и обозначается ключевым словом **default**.

**default** IntegerMath swap() {

**return** (a, b) -> operation(b, a);

}

}

**private** **static** **int** apply(**int** a, **int** b, IntegerMath op) {

**return** op.operation(a, b);

}

**public** **static** **void** main(String... args) {

IntegerMath addition = (a, b) -> a + b;

IntegerMath subtraction = (a, b) -> a - b;

System.***out***.println("40 + 2 = " +

*apply*(40, 2, addition));

System.***out***.println("20 - 10 = " +

*apply*(20, 10, subtraction));

System.***out***.println("10 - 20 = " +

*apply*(20, 10, subtraction.swap()));

}

}

In this example, a functional interface called IntegerMath is declared. Lambda expressions that implement IntegerMath are passed to the apply() method to be executed. Default methods like swap define methods on functions.

Java 8 introduced another mechanism named method reference (the :: operator) to create a lambda on an existing method. A method reference doesn't indicate the number or types of arguments because those are extracted from the abstract method of the functional interface.

IntBinaryOperator sum = Integer::sum;

In the example above, the functional interface IntBinaryOperator declares an abstract method int applyAsInt(int, int), so the compiler looks for a method int sum(int, int) in the class java.lang.Integer.

В Java 8 лямбды имеют следующие ограничения:

* Лямбды могут выбрасывать исключения, поэтому такие лямбды не будут работать с интерфейсами, используемыми в коллекциях (Collection API).
* Переменные из области видимости, где была объявлена лямбда могут быть доступны в лямбде только если они являются эффективно финальными, т.е. остаются неизменными внутри или вне области видимости лямбды. Другими словами они объявлены как final или не изменяются после инициализации.

В следующем примере лямбды передаются в метод в качестве параметра.

**public** **class** Lambda {

**public** **static** **void** main(String[] args) {

Lambda myLambda = **new** Lambda();

BinOp add = (a, b) -> a + b;

BinOp sub = (a, b) -> a - b;

System.***out***.println("7 + 2 = " +

myLambda.binOperation(7, 2, add));//7 + 2 = 9

System.***out***.println("7 - 2 = " +

myLambda.binOperation(7, 2, sub));//7 - 2 = 5

}

**interface** BinOp{

**int** calculate(**int** x, **int** y);

}

**public** **int** binOperation(**int** a, **int** b, BinOp op) {

**return** op.calculate(a, b);

}

}

Чтобы объявить и использовать лямбда-выражение, основная программа разбивается на ряд этапов:

1. Определение функционального интерфейса и ссылки на него:

BinOp operation;

1. Создание лямбда-выражения:

operation = (x,y)->x+y;

Параметры лямбда-выражения соответствуют параметрам метода без реализации интерфейса BinOp, а результат соответствует возвращаемому результату этого метода. Для возврата результата из лямбда-выражения return не используется.

1. Использование лямбда-выражения в виде вызова метода интерфейса:

**int** result = operation.calculate(10, 20);

Для одного функционального интерфейса можно определить множество лямбда-выражений. Например:

BinOp operation1 = (**int** x, **int** y)-> x + y;

BinOp operation2 = (**int** x, **int** y)-> x - y;

BinOp operation3 = (**int** x, **int** y)-> x \* y;

System.***out***.println(operation1.calculate(20, 10)); //30

System.***out***.println(operation2.calculate(20, 10)); //10 System.***out***.println(operation3.calculate(20, 10)); //200

# Отложенное выполнение

Одним из ключевых моментов в использовании лямбд является отложенное выполнение (deferred execution). Определяем в одном месте программы лямбда-выражение и затем можем его вызывать при необходимости неопределенное количество раз в различных частях программы. Отложенное выполнение может потребоваться, к примеру, в следующих случаях:

* Выполнение кода отдельном потоке.
* Выполнение одного и того же кода несколько раз.
* Выполнение кода в результате какого-то события.
* Выполнение кода только в том случае, когда он действительно необходим.

# Передача параметров в лямбда-выражение

Параметры лямбда-выражения должны соответствовать по типу параметрам метода из функционального интерфейса. При написании самого лямбда-выражения тип параметров писать необязательно, хотя в принципе это можно сделать, например:

operation = (int x, int y)->x+y;

Если метод не принимает никаких параметров, то пишутся пустые скобки, например:

()-> 30 + 20;

Если метод принимает только один параметр, то скобки можно опустить:

n-> n \* n;

# Терминальные лямбда-выражения

Выше рассмотрены лямбда-выражения, которые возвращают определенное значение. Но также могут быть и терминальные лямбды, которые не возвращают никакого значения. Например:

**interface** Printable{

**void** print(String s);

}

**public** **class** Lambda {

**public** **static** **void** main(String[] args) {

Printable printer = s->System.***out***.println(s);

printer.print("Hello Java!");

}

}

# Лямбды и локальные переменные

Лямбда-выражение может использовать переменные, которые объявлены во внешней более общей области видимости – на уровне класса или метода, в котором лямбда-выражение определено. Однако в зависимости от того, как и где определены переменные, могут различаться способы их использования в лямбдах. Рассмотрим первый пример – использования переменных уровня класса:

**public** **class** Lambda {

**static** **int** *x* = 10;

**static** **int** *y* = 20;

**public** **static** **void** main(String[] args) {

Operation op = ()->{

*x*=30;

**return** *x*+*y*;

};

System.***out***.println(op.calculate()); // 50

System.***out***.println(*x*); // 30 - значение x изменилось

}

}

**interface** Operation{

**int** calculate();

}

Переменные x и y объявлены на уровне класса, и в лямбда-выражении мы их может получить и даже изменить. Так, в данном случае после выполнения выражения изменяется значение переменной x.

Рассмотрим другой пример - локальные переменные на уровне метода:

**public** **class** Lambda {

**public** **static** **void** main(String[] args) {

**int** n=70;

**int** m=30;

Operation op = ()->{

//n=100; - так нельзя сделать

**return** m+n;

};

// n=100; - так тоже нельзя

System.***out***.println(op.calculate()); // 100

}

**interface** Operation{

**int** calculate();

}

}

Локальные переменные уровня метода мы также может использовать в лямбдах, но нельзя изменять их значение. Эти переменные должны быть финальными или эффективно финальными (final), т.е. их нельзя изменять даже вне лямбда выражений.

# Блоки кода в лямбда-выражениях

Существуют два типа лямбда-выражений: однострочное выражение и блок кода. Примеры однострочных выражений демонстрировались выше. Блочные выражения обрамляются фигурными скобками. В блочных лямбда-выражениях можно использовать внутренние вложенные блоки, циклы, конструкции if, switch, создавать переменные и т.д. Если блочное лямбда-выражение должно возвращать значение, то явным образом применяется оператор return:

Operationable operation = (**int** x, **int** y)-> {

**if**(y==0)

**return** 0;

**else**

**return** x/y;

};

System.out.println(operation.calculate(20, 10)); //2

System.out.println(operation.calculate(20, 0)); //0

# Обобщенный функциональный интерфейс

Функциональный интерфейс может быть обобщенным, однако в лямбда-выражении использование обобщений не допускается. В этом случае нам надо типизировать объект интерфейса определенным типом, который потом будет применяться в лямбда-выражении. Например:

**public** **class** Lambda {

**public** **static** **void** main(String[] args) {

Operationable<Integer> operation1 = (x, y)-> x + y;

Operationable<String> operation2 = (x, y) -> x + y;

System.***out***.println(operation1.calculate(20, 10)); //30

System.***out***.println(operation2.calculate("20", "10")); //2010

}

}

**interface** Operationable<T>{

T calculate(T x, T y);

}

<https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>