Популярно о лямбда-выражениях в Java. С примерами и задачами.

**Для кого предназначена эта статья?**

* для тех, кто считает, что уже неплохо знает Java Core, но понятия не имеет о лямбда-выражениях в Java. Или, возможно, что-то уже слышал про лямбды, но без подробностей.
* для тех, у кого есть какое-то понимание лямбда-выражений, но использовать их до сих пор боязно и непривычно.

Если вы не входите в одну из этих категорий, вам эта статья вполне может показаться скучной, некорректной и вообще «не айс». В таком случае либо смело проходите мимо, либо, если вы хорошо разбираетесь в теме, предложите в комментариях, как я мог бы улучшить или дополнить статью. Материал не претендует на какую-либо академическую ценность, и уж тем-более на новизну. Скорее, наоборот: в ней я попытаюсь описать сложные (для кого-то) вещи как можно проще. На написание меня подвигла просьба объяснить stream api. Я подумал, и решил, что без понимания лямбда-выражений, часть моих примеров о «стримах» будет непонятна. Так что для начала — лямбды. **Какие знания требуются для понимания этой статьи:**

1. Понимание объектно-ориентированного программирования (далее ООП), а именно:
   * знание что такое классы, объекты, какая между ними разница;
   * знание что такое интерфейсы, чем они отличаются от классов, какая между ними (интерфейсами и классами) связь;
   * знание что такое метод, как его вызвать, что такое абстрактный метод (или метод без реализации), что такое параметры/аргументы метода, как их туда передавать;
   * модификаторы доступа, статические методы/переменные, финальные методы/переменные;
   * наследование (классов, интерфейсов, множественное наследование интерфейсов).
2. Знание Java Core: обобщенные типы (generics), коллекции (списки), потоки (threads).

Ну что ж, приступим.

**Немного истории**

Лямбда-выражения пришли в Java из функционального программирования, а туда — из математики. В середине 20го века в Америке в Принстонском университете работал некий Алонзо Чёрч, который очень любил математику и всяческие абстракции. Именно Алонзо Чёрч и придумал лямбда-исчисление, которое поначалу было набором неких абстрактных идей и никак не касалось программирования. В то же время в том же Принстонском университете работали такие математики, как Алан Тьюринг и Джон фон Нейман. Всё сложилось воедино: Чёрч придумал систему лямбда-исчислений, Тьюринг разработал свою абстрактную вычислительную машину, нынче известную под названием «машина Тьюринга». Ну а фон Нейман предложил схему архитектуры вычислительных машин, которая легла в основу современных компьютеров (и сейчас называется «архитектура фон Неймана»). В то время идеи Алонзо Чёрча не обрели столь громкую известность, как работы его коллег (за исключением сферы «чистой» математики). Тем не менее, чуть позже некто Джон МакКарти (также выпускник Принстонского университета, на момент рассказа — сотрудник Массачусетского технологического института) заинтересовался идеями Чёрча. На их основе, в 1958 году он создал первый функциональный язык программирования Lisp. А спустя 58 лет идеи функционального программирования просочились в Java под номером 8. Не прошло и 70 лет…

**Суть**

**Лямбда-выражение** — это такая функция. Можете считать, что это обычный метод в Java, только его особенность в том, что его можно передавать в другие методы в качестве аргумента.

Да, стало возможным передавать в методы не только числа, строки и котиков, но и другие методы!

Когда нам это может понадобиться? Например, если мы хотим передать какой-нибудь callback. Нам нужно, чтобы тот метод, который мы вызываем, имел возможность вызвать какой-то другой метод, который мы ему передадим. То есть, чтобы у нас была возможность в каких-то случаях передавать один callback, а в других — иной. И наш метод, который бы принимал наши callback-и, — вызывал бы их.

Простой пример — сортировка. Допустим, мы пишем какую-то хитрую сортировку, которая выглядит примерно вот так:

**public** **void** mySuperSort() {

// ... тут че-то делаем

**if**(compare(obj1, obj2) > 0)

// ... и тут че-то делаем

}

Там, где if, мы вызываем метод compare(), передаем туда два объекта, которые мы сравниваем, и хотим узнать какой из этих объектов «больше». Тот, который «больше» мы поставим перед тем, который «меньше». Я написал «больше» в кавычках, потому что мы пишем универсальный метод, который будет уметь сортировать не только по возрастанию, но и по убыванию (в таком случае «больше» будет тот объект, который по сути меньше, и наоборот).

Чтобы задать правило, как именно мы хотим отсортировать, нам нужно каким-то образом передать его в наш метод mySuperSort(). В таком случае мы сможем как-то «управлять» нашим методом во время его вызова.

Разумеется, можно написать два отдельных метода mySuperSortAsc() и mySuperSortDesc() для сортировки по возрастанию и убыванию. Или передавать какой-то параметр внутрь метода (допустим boolean и если true, сортировать по возрастанию, а если false — по убыванию).

А что, если мы хотим отсортировать не какую-то простую структуру, а например список массивов строк? Как наш метод mySuperSort() будет знать, по какому принципу сортировать эти массивы строк? По размеру? По общей длине слов? Быть может, по алфавиту, в зависимости от первой строки в массиве? А что, если нам в каких-то случаях надо отсортировать список массивов по размеру массива, а в другом случае — по суммарной длине слов в массиве?

Я думаю, вы уже слышали о компараторах и о том, что в таких случаях мы просто передаем в наш метод сортировки объект компаратора, в котором мы и описываем правила, по которым мы хотим сортировать.

Поскольку стандартный метод sort() реализован по тому же принципу, что и mySuperSort(), в примерах я буду использовать именно стандартный sort().

String[] array1 = {"мама", "мыла", "раму"};

String[] array2 = {"я", "очень", "люблю", "java"};

String[] array3 = {"мир", "труд", "май"};

List<String[]> arrays = **new** ArrayList<>();

arrays.add(array1);

arrays.add(array2);

arrays.add(array3);

Comparator<String[]> sortByLength = **new** Comparator<String[]>() {

@Override

**public** **int** compare(String[] o1, String[] o2) {

**return** o1.length - o2.length;

}

};

Comparator<String[]> sortByWordsLength = **new** Comparator<String[]>() {

@Override

**public** **int** compare(String[] o1, String[] o2) {

**int** length1 = 0;

**int** length2 = 0;

**for** (String s : o1) {

length1 += s.length();

}

**for** (String s : o2) {

length2 += s.length();

}

**return** length1 - length2;

}

};

arrays.sort(sortByLength);

**Результат:**

1. мама мыла раму
2. мир труд май
3. я очень люблю java

Тут массивы отсортированы по количеству слов в каждом массиве. Массив, где меньше слов — тот и считается «меньшим». Вот почему он стоит в начале. Тот, где слов больше считается «больше», и оказывается в конце.

Если в метод sort() мы передадим другой компаратор (sortByWordsLength) — то и **результат** будет другой:

1. мир труд май
2. мама мыла раму
3. я очень люблю java

Теперь массивы отсортированы по общему количеству букв в словах такого массива. В первом случае 10 букв, во втором 12, и в третьем 15.

Если у нас используется только один компаратор, то мы можем не заводить под него отдельную переменную, а просто создать объект анонимного класса прямо в момент вызова метода sort(). Примерно так:

String[] array1 = {"мама", "мыла", "раму"};

String[] array2 = {"я", "очень", "люблю", "java"};

String[] array3 = {"мир", "труд", "май"};

List<String[]> arrays = **new** ArrayList<>();

arrays.add(array1);

arrays.add(array2);

arrays.add(array3);

arrays.sort(**new** Comparator<String[]>() {

@Override

**public** **int** compare(String[] o1, String[] o2) {

**return** o1.length - o2.length;

}

});

**Результат будет такой же, как и в первом случае.**

**Задача 1**. Переписать этот пример так, чтобы он сортировал массивы не по возрастанию количества слов в массиве, а по убыванию.

Это всё мы уже знаем. Мы умеем передавать объекты в методы, мы можем передать тот или иной объект в метод в зависимости от того, что нам в данный момент надо, и внутри того метода, куда мы передаем такой объект, будет вызван тот метод, для которого мы написали реализацию. Возникает вопрос: при чём здесь вообще лямбда-выражения?

**При том, что лямбда — это и есть такой объект, который содержит ровно один метод. Такой себе объект-метод. Метод, запакованный в объект. Просто у них немного непривычный синтаксис (но об этом чуть позже).** **Давайте еще раз взглянем на эту запись**

arrays.sort(**new** Comparator<String[]>() {

@Override

**public** **int** compare(String[] o1, String[] o2) {

**return** o1.length - o2.length;

}

});

Тут мы берем наш список arrays и вызываем у него метод sort(), куда передаем объект компаратора с одним единственным методом compare() (нам не важно, как он называется, ведь он — единственный в этом объекте, тут не промахнемся). Этот метод принимает два параметра, с которыми мы дальше и работаем.

Если вы работаете в **IntelliJ IDEA**, то наверняка видели, как она вам предлагает этот код значительно сократить:

arrays.sort((o1, o2) -> o1.length - o2.length);

Вот так шесть строк превратились в одну короткую. 6 строк переписали в одну короткую. Что-то исчезло, но я гарантирую, что не исчезло ничего важного, и такой код будет работать абсолютно так же, как и при анонимном классе.

**Задача 2**. Догадаться, как переписать решение задачи 1 через лямбды (в крайнем случае, попросите **IntelliJ IDEA** превратить ваш анонимный класс в лямбду).

**Поговорим об интерфейсах**

В принципе, интерфейс — это просто список абстрактных методов. Когда мы создаем класс и говорим, что он будет имплементировать какой-то интерфейс — мы должны в нашем классе написать реализацию тех методов, которые перечислены в интерфейсе (или, на крайний случай, не писать, но сделать класс абстрактным). Бывают интерфейсы со множеством разных методов (например List), бывают интерфейсы только с одним методом (например, тот же Comparator или Runnable).

Бывают интерфейсы и вовсе без единого метода (так называемые интерфейсы-маркеры, например Serializable). Те интерфейсы, у которых только один метод, также называют **функциональными интерфейсами**. В Java 8 они даже помечены специальной аннотацией **@FunctionalInterface**.

Именно интерфейсы с одним единственным методом и подходят для использования лямбда-выражениями. Как я уже говорил выше, лямбда-выражение — это метод, завернутый в объект. И когда мы передаем куда-то такой объект — мы, по сути, передаем этот один единственный метод. Получается, нам не важно, как этот метод называется. Все, что нам важно — это параметры, которые этот метод принимает, и, собственно, сам код метода.

Лямбда-выражение — это, по сути. реализация функционального интерфейса. Где видим интерфейс с одним методом — значит, такой анонимный класс можем переписать через лямбду. Если в интерфейсе больше/меньше одного метода — тогда нам лямбда-выражение не подойдет, и будем использовать анонимный класс, или даже обычный.

Пришло время поковырять лямбды. :)

**Синтаксис**

Общий синтаксис примерно такой:

(параметры) -> {тело метода}

То есть, круглые скобки, внутри их параметры метода, «стрелочка» (это два символа подряд: минус и больше), после которой тело метода в фигурных скобках, как и всегда. Параметры соответствуют тем, что указаны в интерфейсе при описании метода. Если тип переменных может быть четко определен компилятором (в нашем случае, точно известно, что мы работаем с массивами строк, потому что List — типизирован именно массивами строк), то и тип переменных String[] можно не писать.

|  |
| --- |
| Если не уверены, указывайте тип, а IDEA подсветит его серым, если он не нужен. |

Подробнее можно почитать в [туториале оракла](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#target-typing), например. Это называется **«target typing»**. Имена переменным можно дать какие угодно, не обязательно именно те, которые указаны в интерфейсе.

Если параметров нет — тогда просто круглые скобки. Если параметр только один —просто имя переменной без круглых скобок.

С параметрами разобрались, теперь про тело самого лямбда-выражения. Внутри фигурных скобок пишете код, как для обычного метода. Если у вас весь код состоит только из одной строки, можете вообще фигурных скобок не писать (как и с if-ами, и с циклами).

Если же ваша лямбда что-то возвращает, но ее тело состоит из одной строки, писать return вовсе не обязательно. А вот если у вас фигурные скобки, тогда, как и в обычном методе, нужно явно писать return.

**Примеры**

**Пример 1.**

() -> {}

Самый простой вариант. И самый бессмысленный:).Так как ничего не делает.

**Пример 2.**

() -> ""

Тоже интересный вариант. Ничего не принимает и возвращает пустую строку (return опущен за ненадобностью). То же, но с return:

() -> {

**return** "";

}

**Пример 3. Hello world на лямбдах**

() -> System.out.println("Hello world!")

Ничего не принимает, ничего не возвращает (мы не можем поставить return перед вызовом System.out.println(), так как тип возвращаемого значения в методе println() — void), просто выводит на экран надпись. Идеально подходит для реализации интерфейса Runnable. Этот же пример более полный:

**public** **class** Main {

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

**new** Thread(() -> System.out.println("Hello world!")).start();

}

}

Ну, или так:

**public** **class** Main {

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

Thread t = **new** Thread(() -> System.out.println("Hello world!"));

t.start();

}

}

Или даже можем сохранить лямбда-выражение как объект типа Runnable, а потом его уже передать в конструктор thread’а:

**public** **class** Main {

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

Runnable runnable = () -> System.out.println("Hello world!");

Thread t = **new** Thread(runnable);

t.start();

}

}

Рассмотрим подробнее момент сохранения лямбда-выражения в переменную. Интерфейс Runnable нам говорит, что его объекты должны иметь метод public void run().

Согласно интерфейсу, метод run ничего не принимает в качестве параметров. И ничего не возвращает (void). Поэтому при такой записи будет создан объект с каким-то методом, который ничего не принимает и не возвращает. Что вполне соответствует методу run() в интерфейсе Runnable. Вот почему мы и смогли поместить это лямбда-выражение в переменную типа Runnable.

**Пример 4**

() -> 42

Снова, ничего не принимает, а возвращает число 42.

Такое лямбда-выражение можно поместить в переменную типа Callable, потому что в этом интерфейсе определен только один метод, который выглядит примерно так:

V call(),

где V — это тип возвращаемого значения (в нашем случае int).

Соответственно, мы можем сохранить такое лямбда-выражение следующим образом:

Callable<Integer> c = () -> 42;

**Пример 5. Лямбда в несколько строк**

() -> {

String[] helloWorld = {"Hello", "world!"};

System.out.println(helloWorld[0]);

System.out.println(helloWorld[1]);

}

Опять, это лямбда-выражение без параметров и тип возвращаемого значения у него void (так как отсутствует return).

**Пример 6**

x -> x

Тут мы принимаем что-то в переменную х, и ее же и возвращаем. Обратите внимание, что если принимается только один параметр — то скобки вокруг него можно не писать.

То же, но со скобками:

(x) -> x

А вот вариант с явным return:

x -> {

**return** x;

}

Или так, со скобками и return:

(x) -> {

**return** x;

}

Или с явным указанием типа (и, соответственно, со скобками):

(**int** x) -> x

**Пример 7**

x -> ++x

Принимаем х, возвращаем его же, но на 1 больше. Можно переписать и так:

x -> x + 1

В обоих случаях скобки вокруг параметра, тела метода и слово return не указываем, так как это не обязательно.

Варианты со скобками и с ретурном описаны в примере 6.

**Пример 8**

(x, y) -> x % y

Принимаем какие-то х и у, возвращаем остаток от деления x на y. Скобки вокруг параметров тут уже обязательны. Необязательны они только когда параметр всего один. Вот так с явным указанием типов:

(**double** x, **int** y) -> x % y

**Пример 9**

(Cat cat, String name, **int** age) -> {

cat.setName(name);

cat.setAge(age);

}

Принимаем объект Кот, строку с именем и целое число возраст. В самом методе устанавливаем Коту переданные имя и возраст.

Поскольку переменная cat у нас ссылочного типа, то и объект Кот вне лямбда-выражения изменится (получит переданные внутрь имя и возраст). Немного усложненный вариант, где используется подобная лямбда:

**public** **class** Main {

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

// создаем кота и выводим на экран чтоб убедиться, что он "пустой"

Cat myCat = **new** Cat();

System.out.println(myCat);

// создаем лямбду

Settable<Cat> s = (obj, name, age) -> {

obj.setName(name);

obj.setAge(age);

};

// вызываем метод, в который передаем кота и лямбду

changeEntity(myCat, s);

// выводим на экран и видим, что состояние кота изменилось (имеет имя и возраст)

System.out.println(myCat);

}

**private** **static** <T **extends** WithNameAndAge> **void** changeEntity(T entity, Settable<T> s) {

s.set(entity, "Мурзик", 3);

}

}

**interface** WithNameAndAge {

**void** setName(String name);

**void** setAge(**int** age);

}

**interface** Settable<C **extends** WithNameAndAge> {

**void** set(C entity, String name, **int** age);

}

**class** Cat **implements** WithNameAndAge {

**private** String name;

**private** **int** age;

@Override

**public** **void** setName(String name) {

**this**.name = name;

}

@Override

**public** **void** setAge(**int** age) {

**this**.age = age;

}

@Override

**public** String toString() {

**return** "Cat{" +

"name='" + name + '\'' +

", age=" + age +

'}';

}

}

**Результат:** ***Cat{name='null', age=0} Cat{name='Мурзик', age=3}***

Как видно, сначала объект Кот имел одно состояние, а после использования лямбда-выражения, состояние изменилось.

Лямбда-выражения отлично сочетаются с дженериками. И если нам понадобится создать класс Dog, например, который тоже будет имплементить WithNameAndAge, то в методе main() мы можем те же операции проделать и с Cобакой, абсолютно не меняя сами лямбда-выражение.

**Задача 3**. Написать функциональный интерфейс с методом, который принимает число и возвращает булево значение. Написать реализацию такого интерфейса в виде лямбда-выражения, которое возвращает true если переданное число делится без остатка на 13.

**Задача 4**. Написать функциональный интерфейс с методом, который принимает две строки и возвращает тоже строку. Написать реализацию такого интерфейса в виде лямбды, которая возвращает ту строку, которая длиннее.

**Задача 5**. Написать функциональный интерфейс с методом, который принимает три дробных числа: a, b, c и возвращает тоже дробное число. Написать реализацию такого интерфейса в виде лямбда-выражения, которое возвращает дискриминант.

Кто забыл, **D = b^2 — 4ac**.

**Задача 6**. Используя функциональный интерфейс из задачи 5 написать лямбда-выражение, которое возвращает результат операции a \* b^c.

**Доступ к внешним переменным**

Скомпилируется ли такой код с анонимным классом?

**int** counter = 0;

Runnable r = **new** Runnable() {

@Override

**public** **void** run() {

counter++;

}

};

Нет. Переменная counter должна быть final. Или не обязательно final, но в любом случае изменять свое значение она не может.

Тот же принцип используется и в лямбда-выражениях. Они имеют доступ ко всем переменным, которые им «видны» из того места, где они объявлены. Но лямбда не должна их изменять (присваивать новое значение).

Правда, есть вариант обхода этого ограничения в анонимных классах. Достаточно лишь создать переменную ссылочного типа и менять внутреннее состояние объекта. При этом сама переменная будет указывать на тот же объект, и в таком случае можно смело указывать её как final.

**final** AtomicInteger counter = **new** AtomicInteger(0);

Runnable r = **new** Runnable() {

@Override

**public** **void** run() {

counter.incrementAndGet();

}

};

Здесь у нас переменная counter является ссылкой на объект типа AtomicInteger. А для изменения состояния этого объекта используется метод incrementAndGet(). Значение самой переменной во время работы программы не меняется и всегда указывает на один и тот же объект, что позволяет нам объявить переменную сразу с ключевым словом final.

Эти же примеры, но с лямбда-выражениями:

**int** counter = 0;

Runnable r = () -> counter++;

Не скомпилируется по той же причине, что и вариант с анонимным классом: counter не должна меняться во время работы программы.

Зато вот так — все нормально:

**final** AtomicInteger counter = **new** AtomicInteger(0);

Runnable r = () -> counter.incrementAndGet();

Это касается и вызова методов. Изнутри лямбда-выражения можно не только обращаться ко всем «видимым» переменным, но и вызывать те методы, к которым есть доступ.

**public** **class** Main {

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

Runnable runnable = () -> staticMethod();

**new** Thread(runnable).start();

}

**private** **static** **void** staticMethod() {

System.out.println("Я - метод staticMethod(), и меня только-что кто-то вызвал!");

}

}

Хотя метод staticMethod() и приватный, но он «доступен» для вызова внутри метода main(), поэтому точно так же доступен для вызова изнутри лямбды, которая создана в методе main.

**Момент выполнения кода лямбда-выражения**

Возможно, вам этот вопрос покажется слишком простым, но его всё следует задать: когда выполнится код внутри лямбда-выражения? В момент создания? Или же в тот момент, когда (еще и неизвестно где) оно будет вызвано? Проверить довольно просто.

System.out.println("Запуск программы");

// много всякого разного кода

// ...

System.out.println("Перед объявлением лямбды");

Runnable runnable = () -> System.out.println("Я - лямбда!");

System.out.println("После объявления лямбды");

// много всякого другого кода

// ...

System.out.println("Перед передачей лямбды в тред");

**new** Thread(runnable).start();

Вывод на экран:

Запуск программы

Перед объявлением лямбды

После объявления лямбды

Перед передачей лямбды в тред

Я - лямбда!

Видно, что код лямбда-выражения выполнился в самом конце, после того, как был создан тред и лишь когда процесс выполнения программы дошел до фактического выполнения метода run(). А вовсе не в момент его объявления. Объявив лямбда-выражение, мы лишь создали объект типа Runnable и описали поведение его метода run(). Сам же метод был запущен значительно позже.

**Method References (Ссылки на методы)?**

Не имеет прямого отношения к самим лямбдам, но я считаю, что будет логично сказать об этом пару слов в этой статье.

Допустим, у нас есть лямбда-выражение, которое не делает ничего особенного, а просто вызывает какой-то метод.

x -> System.out.println(x)

Ему передали некий х, а оно — просто вызвало System.out.println() и передало туда х.

В таком случае, мы можем заменить его на ссылку на нужный нам метод. Вот так:

System.out::println

Да, без скобок в конце! Более полный пример:

List<String> strings = **new** LinkedList<>();

strings.add("мама");

strings.add("мыла");

strings.add("раму");

strings.forEach(x -> System.out.println(x));

В последней строке мы используем метод forEach(), который принимает объект интерфейса Consumer. Это снова же функциональный интерфейс, у которого только один метод void accept(T t). Соответственно, мы пишем лямбда-выражение, которое принимает один параметр (поскольку он типизирован в самом интерфейсе, тип параметра мы не указываем, а указываем, что называться он у нас будет х). В теле лямбда-выражения пишем код, который будет выполняться при вызове метода accept().

Здесь мы просто выводим на экран то, что попало в переменную х. Сам же метод forEach() проходит по всем элементам коллекции, вызывает у переданного ему объекта интерфейса Consumer (нашей лямбды) метод accept(), куда и передает каждый элемент из коллекции.

Как я уже сказал, такое лямбда-выражение (просто вызывающее другой метод) мы можем заменить ссылкой на нужный нам метод. Тогда наш код будет выглядеть так:

List<String> strings = **new** LinkedList<>();

strings.add("мама");

strings.add("мыла");

strings.add("раму");

strings.forEach(System.out::println);

Главное, чтобы совпадали принимаемые параметры методов (println() и accept()). Поскольку метод println() может принимать что угодно (он перегружен для всех примитивов и для любых объектов, мы можем вместо лямбда-выражения передать в forEach() просто ссылку на метод println(). Тогда forEach() будет брать каждый элемент коллекции и передавать его напрямую в метод println().

Кто сталкивается с этим впервые, обратите внимание, что мы не вызываем метод System.out.println() (с точками между словами и со скобочками в конце), а именно передаем саму ссылку на этот метод. При такой записи

strings.forEach(System.out.println());

у нас будет ошибка компиляции. Поскольку перед вызовом forEach() Java увидит, что вызывается System.out.println(), поймет, что возвращается void и будет пытаться этот void передать в forEach(), который там ждет объект типа Consumer.

**Синтаксис использования Method References**

Он довольно прост:

1. Передаем ссылку на статический метод ИмяКласса:: имяСтатическогоМетода?

1. **public** **class** Main {
2. **public** **static** **void** main(String[] args) {
3. List<String> strings = **new** LinkedList<>();
4. strings.add("мама");
5. strings.add("мыла");
6. strings.add("раму");
7. strings.forEach(Main::staticMethod);
8. }
9. **private** **static** **void** staticMethod(String s) {
10. // do something
11. }

}

2. Передаем ссылку на не статический метод используя существующий объект имяПеременнойСОбъектом:: имяМетода

* 1. **public** **class** Main {

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

1. List<String> strings = **new** LinkedList<>();
2. strings.add("мама");
3. strings.add("мыла");
4. strings.add("раму");
5. Main instance = **new** Main();
6. strings.forEach(instance::nonStaticMethod);
7. }
8. **private** **void** nonStaticMethod(String s) {
9. // do something
10. }

}

3. Передаем ссылку на не статический метод используя класс, в котором реализован такой метод ИмяКласса:: имяМетода

1. **public** **class** Main {
2. **public** **static** **void** main(String[] args) {
3. List<User> users = **new** LinkedList<>();
4. users.add(**new** User("Вася"));
5. users.add(**new** User("Коля"));
6. users.add(**new** User("Петя"));
7. users.forEach(User::print);
8. }
9. **private** **static** **class** User {
10. **private** String name;
11. **private** User(String name) {
12. **this**.name = name;
13. }
14. **private** **void** print() {
15. System.out.println(name);
16. }
17. }
18. }

4. Передаем ссылку на конструктор ИмяКласса::new

Использование ссылок на методы очень удобно, когда есть готовый метод , который вас полностью устраивает, и вы бы хотели использовать его в качестве callback-а. В таком случае, вместо того чтобы писать лямбда-выражение с кодом того метода, или же лямбда-выражение, где мы просто вызываем этот метод, мы просто передаем ссылку на него. И всё.

**Интересное различие между анонимным классом и лямбда-выражением**

В анонимном классе ключевое слово this указывает на объект этого анонимного класса. А если использовать this внутри лямбды, мы получим доступ к объекту обрамляющего класса. Того, где мы это выражение, собственно, и написали. Так происходит потому, что лямбда-выражения при компиляции превращаются в приватный метод того класса, где они написаны. Использовать эту «фичу» я бы не рекомендовал, поскольку у неё есть побочный эффект (side effect), что противоречит принципам функционального программирования. Зато такой подход вполне соответствует ООП. ;)