**Лямбда-выражения** – экземпляр функционального интерфейса.

\* Попытка привнести в Java элементы функционального программирования, а именно:

Функция – объект первого класса: функции могут присваиваться переменным, передаваться в качестве аргументов, возвращаться из других функций (основная).

**Синтаксис:**

Структура выражения – (Аргументы метода) -> {Тело метода}

Аргументы:

могут быть записаны без явного указания их типов.  
Если аргумент один и не указан его тип, то скобки могут быть опущены.

Тело метода:

Блок кода – стандартное тело метода (можно объявлять локальные переменные, циклы, операторы и др.) – синтаксис включает фигурные скобки и оператор return (если требуется возврат значения).

Выражение – тело выражено единственным выражением или возвращаемым значением – можно опустить return и фигурные скобки.

**Свойства:**

1. Реализация функционального интерфейса
2. Не имеет сигнатуры при объявлении. По объявленному выражению невозможно понять тип, он определяется по контексту
3. Не имеет состояния
4. Отложенное выполнение

1) Компактная реализация функционального интерфейса – Java не функциональный язык и каждая «функция» должна принадлежать какому-то объектному типу. В качестве таких типов выступают функциональные интерфейсы. То есть структурно лямбда выражение представляет из себя создание некого класса (с неопределенным именем), который будет имплементировать этот интерфейс и реализовывать логику единственного «абстрактного» метода.

Функциональный интерфейс (Single abstract method - SAM) – интерфейс или абстрактный класс, имеющий только один абстрактный метод.

**Лямбда VS анонимный класс.**

Является частным случаем (упрощенной версией) анонимного класса:

Включает только реализованный абстрактный метод - нельзя объявлять новые методы, переменные класса. В остальном обладает свойствами анонимного класса: может использовать переменные и методы обрамляющего класса независимо от модификатора доступа (только final и effective-final), обладает свойствами замыкания и отложенного выполнения. В Отличие от анонимного класса лямбда не компилируется в полноценный класс (компилируется в синтетический метод класса, в котором была создана, а в местах ее вызова вместо обычного вызова метода используется инструкция invokedynamic)

Ограничение на использование final и effective-final переменных можно обойти путем использования переменных ссылочного типа и изменения их значений (не лучшая практика).

Лямбда: более компактная/читаемая запись, оптимизирована компилятором, но не может использоваться как полноценный класс.  
Анонимный класс: предоставляет возможности обычного класса (методы, поля, конструкторы). Соответственно незаменима в случае, когда нужен функционал класса (объявить несколько методов или переменные в коде).

Лямбда и анонимные классы являются взаимозаменяемыми конструкциями, но лямбда — это не анонимный класс на уровне компилятора, анонимный класс требует больше ресурсов для компиляции.

2) Является анонимной функцией – само по себе лямбда выражение не содержит сигнатуры метода, его тип определяется по контексту, в котором оно используется (контекстом является тип переменной или ожидаемое значение метода, в который передается выражение) и по этому типу определяется соответствие ожидаемой функции и фактической функции.

3) Не имеет собственного изменяемого состояния, но может захватывать состояния внешнего контекста.

**Замыкание (Closure)** – совокупность лямбда-выражения или анонимного класса и его захваченных переменных.

**Захват переменных (Capture)** – Сохранение копии данных из области видимости функции локального класса или лямбда выражения. Необходим, т.к. лямбда функция может существовать дольше ее контекста (метод, в котором создавалась лямбда был завершен, и его локальная переменная была стерта).

Локальные переменные - захвачены могут быть только final и effective-final – чтобы не было путаницы с захваченными значениями (какое именно значение захватится), захват происходит вокруг значения – это связано с гарантией языка о том, что локальная переменная не может быть изменена извне (только явно в ее методе).

Поле класса – захват идет по значению this (захват ссылки на объект переменной).

Static поле – не захватывается, так как всегда доступно из контекста (поле читается во время выполнения программы). При этом ссылка на метод захватывает значение static переменной при инициализации метода.

Захват static и non-static полей таким образом приводит к тому, что изменение переменных между созданием и вызовом лямбды приводит к изменению захваченной переменной.

static int *staticField* = 10;  
int field = 15;  
  
public static void main(String[] args) {  
 Main main = new Main();  
 main.foo();  
}  
public void foo() {  
 Runnable run = () -> System.*out*.println(this.field);  
 Runnable run2 = () -> System.*out*.println(*staticField*);  
 field = 121;  
 *staticField* = 122;  
 run.run();  
 run2.run();  
}  
//Изменение static и non-static переменных  
//в интервале между созданием и вызовом функции

\*Ссылка на метод захватывает static переменную во время инициализации.

Consumer<Object> lambda = obj -> System.*out*.println(obj);//не захватывает статик поля, обращается в момент выполнения  
Consumer<Object> ref = System.*out*::println;//захватывает значение статического поля в момент инициализации  
System.*setOut*(null);  
ref.accept("Hello");//отработает  
lambda.accept("Hello");//ошибка, т.к. поле было изменено

Компиляция:

Захваченные переменные преобразуются в поля класса и конфигурируется новый метод, где эти поля передаются в качестве аргументов реализованному функциональному методу.

Лямбда-выражения с преобразованием их в private-методы класса. При этом используется инструкция invokedynamic, появившаяся в Java 7 для динамической привязки метода.

4) Отложенное выполнение (deferred execution) – код внутри лямбда выражения выполняется не в момент инициализации объекта, а в момент вызова соответствующего метода функционального интерфейса (Такая же ситуация при передаче лямбды в качестве аргумента метода).

public static void main(String[] args) {  
 Consumer<Integer> consumer = integer -> System.*out*.println("Выражение выполнено");  
 //Инициализация выражения, но код выполнится только в момент вызова у Consumer его метода accept()  
 System.*out*.println("Не выполнено");  
 *foo*(consumer, 123);}  
public static void foo(Consumer consumer, int num){  
 System.*out*.println("Отрабатывает метод foo");  
 consumer.accept(num); //Только здесь вызовется определённая лямбда функция  
 System.*out*.println("Метод foo отработал");  
}

Не выполнено

Отрабатывает метод foo

Выражение выполнено

Метод foo отработал

**Преимущества отложенного выполнения:**

OrElse(T t) – принимает значение, что требует его вычисления перед вызовом, в случае, если передается метод.

OrElseGet(Supplier sup) – принимает функцию, которая отработает только при явном вызове.

В примере с orElse вычисленное значение метода не будет использовано, но оно вычисляется до захода в метод, OrElseGet исполнит лямбду только если это будет необходимо.

String lineNull = "1.1";  
  
 Double dd = Optional.*ofNullable*(lineNull).map(Double::*parseDouble*).orElse(*getDouble*());  
 Double dddd = Optional.*ofNullable*(lineNull).map(Double::*parseDouble*).orElseGet(Main::*getDouble*);  
}  
public static double getDouble() {  
 System.*out*.println("Отработал getDouble");  
 return 1.1;  
}

**Ссылка на метод** *–* форма записи лямбда выражения, возможно в случае, если тело лямбда функции состоит только из возвращаемого значения и аргументы лямбда функции соответствуют вызываемому методу.

Пример:

Comparator<Integer> comp = (a, b) –> Integer.compare(a, b) – в выражении передаются 2 аргумента типа int и вызываемый метод принимает 2 аргумента типа int и тело состоит только из вызова этого метода => Integer::compare;

**Варианты ссылок:**

BiFunction<Integer, Integer, Integer> staticLambda = (a, b) -> Integer.*sum*(a, b);  
BiFunction<Integer, Integer, Integer> staticRef = Integer::*sum*;  
//Статический метод  
Function<String, String> nonStaticLambda = s -> s.trim();  
Function<String, String> nonStaticRef = String::trim;  
//Нестатический метод  
Predicate<String> nonStaticWithInstanceLambda = anObject -> "foo".equals(anObject);  
Predicate<String> nonStaticWithInstanceRef = "foo"::equals;  
//Нестатический метод с экземпляром  
Supplier<List<String>> constrLambda = () -> new ArrayList<>();  
Supplier<List<String>> constrRef = ArrayList::new;  
//Конструктор

**Разница в обращении к полям в анонимном классе и лямбда выражении:**

Анонимный.

public class Example {  
 int a = 1; //поле обрамляющего класса  
  
 public static void main(String[] args) {  
 Example example = new Example();  
 example.doSmth();  
 }  
 void doSmth() {  
 int b = 2;//локальная переменная метода  
  
 Runnable runnable = new Runnable() {  
 int c = 3; //поле локального класса (анонимного)  
 @Override  
 public void run() {  
 System.*out*.println(Example.this.a);//доступ к полю обрамляющего класса  
 System.*out*.println(b);//доступ к переменной среды (метода)  
 System.*out*.println(this.c); //доступ к полю анонимного класса  
 }  
 };  
 runnable.run();  
 }  
}

Лямбда.

public class Example {  
 int a = 1; //поле обрамляющего класса  
  
 public static void main(String[] args) {  
 Example example = new Example();  
 example.doSmth();  
 }  
 void doSmth() {  
 int b = 2;//локальная переменная метода  
  
 Runnable runnable = ()-> {  
 int c = 3; //переменная МЕТОДА run()  
  
 System.*out*.println(Example.this.a);//доступ к полю обрамляющего класса  
 System.*out*.println(this.a);//Example можно не писать  
 System.*out*.println(b);//доступ к переменной среды (метода)  
 System.*out*.println(c); //доступ к собственной переменной  
 };  
 runnable.run();  
 }  
}

То есть параметр this анонимного класса, указывает на поле, определенное в самом анонимном классе, this лямбда функции указывает на поле обрамляющего класса, поскольку синтаксически невозможно объявить поле класса лямбда выражения.

**Функциональные интерфейсы:**

Пакет java.util.function – пакет, определяющий набор стандартных функциональных интерфейсов с различными комбинациями входных и выходных параметров.

Всего их 42шт присутствуют как Дженерик-интерфейсы, основных 4:

Function: Принимает один аргумент и возвращает результат.

Consumer: Принимает один аргумент, не возвращает результат.

Supplier: Не принимает аргументов, но возвращает результат.

Predicate: Принимает один аргумент и возвращает булево значение

Потоки примитивных типов используют специальные функциональные интерфейсы типа IntFunction, intConsumer и т.д.

[Спека по функциональным интерфейсам](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html)