**Лямбда-выражения Java**

**Введение в лямбда-выражения**

**Последнее обновление: 25.04.2018**

Среди новшеств, которые были привнесены в язык Java с выходом JDK 8, особняком стоят лямбда-выражения. Лямбда представляет набор инструкций, которые можно выделить в отдельную переменную и затем многократно вызвать в различных местах программы.

Есть также ссылки на методы, это таже лямбда, но короче и хуже читаемый код.

Основу лямбда-выражения составляет лямбда-оператор, который представляет стрелку **->**. Этот оператор разделяет лямбда-выражение на две части: левая часть содержит список параметров выражения, а правая собственно представляет тело лямбда-выражения, где выполняются все действия.

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 единственный метод без реализации.

Рассмотрим пример:

|  |  |
| --- | --- |
| 1  2  4  5  6  7  8  9  10  11  12  13  14  15 | public class LambdaApp {       public static void main(String[] args) {  **Operationable op = new Operationable(){**  создаем реализацию метода интерфейса  **public int calculate(int x, int y){**  **return x + y;**              }          };           int z = op.calculate(20, 10);  применяем реализованный метод           System.out.println(z); // 30      }  }  interface Operationable{      int calculate(int x, int y);  } |

По факту лямбда-выражения являются в некотором роде сокращенной формой внутренних анонимных классов, которые ранее применялись в Java. В частности, предыдущий пример мы можем переписать следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class LambdaApp {        public static void main(String[] args) {  создаем реализацию метода интерфейса  **Operationable operation = (x,y)->x+y;**          int result = operation.calculate(10, 20);  применяем реализованный метод          System.out.println(result); //30      }  }  interface Operationable{      int calculate(int x, int y);  } |

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

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

1. Определение ссылки на функциональный интерфейс:

Operationable operation;

2. Создание лямбда-выражения:

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

3. Причем параметры лямбда-выражения соответствуют параметрам единственного метода интерфейса Operationable, а результат соответствует возвращаемому результату метода интерфейса. При этом нам не надо использовать ключевое слово return для возврата результата из лямбда-выражения, написанного одной строкой.

4. Так, в методе интерфейса оба параметра представляют тип int, значит, в теле лямбда-выражения мы можем применить к ним сложение. Результат сложения также представляет тип int, объект которого возвращается методом интерфейса.

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

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

6. Так как в лямбда-выражении определена операция сложения параметров, результатом метода будет сумма чисел 10 и 20.

При этом для одного функционального интерфейса мы можем определить множество лямбда-выражений. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Operationable operation1 = (int x, int y)-> x + y;  Operationable operation2 = (int x, int y)-> x - y;  Operationable 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). То есть мы 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 в 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 и если он необходим

**Передача параметров в лямбда-выражение**

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

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

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

1 ( )-> 30 + 20;

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

1 n-> n \* n;

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.

**Терминальные лямбда-выражения**

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | interface Printable{      void print(String s);  }    public class LambdaApp {      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 вне в 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | interface Operation{      int calculate();  }  public class LambdaApp {        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      }  } |

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

Теперь рассмотрим другой пример - локальные переменные на уровне метода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | 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  } |

Локальные переменные уровня метода мы также может использовать в лямбдах, но изменять их значение мы уже не сможем. Если мы попробуем это сделать, то среда разработки (Netbeans) может нам высветить ошибку и то, что такую переменную надо пометить с помощью ключевого слова final, то есть сделать константой: final int n=70;. Однако это необязательно.

Более того, мы не сможем изменить значение переменной, которая используется в лямбда-выражении, вне этого выражения. То есть даже если такая переменная не объявлена как константа, по сути она является константой.

**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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | 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:

interface Operationable<T>{

    T calculate(T x, T y);

}

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class LambdaApp {        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      }  } |

Таким образом, при объявлении лямбда-выражения ему уже известно, какой тип параметры будут представлять и какой тип они будут возвращать.

**Можно ли заменить каждый анонимный класс выражением лямбда?**

Ответ - нет. Вы можете создать анонимный класс для не заключительных классов и интерфейсов. Не то же самое для лямбда-выражений. Они могут использоваться только там, где ожидается интерфейс SAM, т.е. Интерфейсы только с одним абстрактным методом (до Java 8 каждый метод интерфейса был абстрактным, но поскольку интерфейсы Java 8 могут также иметь стандартные и статические методы, которые не являются абстрактными, поскольку они имеют реализацию).

**Итак, какие анонимные классы можно заменить выражением лямбда?**

Только анонимные классы, которые являются реализациями интерфейса SAM (например, Runnable, ActionListener, Comparator, Predicate), могут быть заменены выражением лямбда. DefaultConsumer не может быть лямбда-мишенью, потому что это даже не интерфейс.

**А как насчет класса** [Consumer](https://www.rabbitmq.com/releases/rabbitmq-java-client/current-javadoc/com/rabbitmq/client/Consumer.html)**?**

Несмотря на то, что [Consumer](https://www.rabbitmq.com/releases/rabbitmq-java-client/current-javadoc/com/rabbitmq/client/Consumer.html) является интерфейсом, он не является интерфейсом SAM, потому что он содержит более одного абстрактного метода.