БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ
Кафедра веб-технологий и компьютерного моделирования

А. С. Кравчук, А. И. Кравчук, Е. В. Кремень

ЯЗЫК JAVA

ЛЯМБДА-ВЫРАЖЕНИЯ

Учебные материалы
для студентов специальности 1-31 03 08
«Математика и информационные технологии
(по направлениям)»

МИНСК
2023

УДК 004.432.045:004.738.5Java (075.8)
ББК 32.973.2-018.1я73-1
К78

Рекомендовано советом
механико-математического факультета БГУ
26 января 2023 г., протокол № 5

Рецензент
кандидат технических наук, доцент *М. Н. Садовская*

Кравчук, А. С.

К78 Язык Java. Лямбда-выражения : учеб. материалы для студентов спец. 1-31 03 08 «Математика и информационные технологии (по направлениям)» / А. С. Кравчук, А. И. Кравчук, Е. В. Кремень. – Минск : БГУ, 2023. – 37 с.

Рассматриваются лямбда-выражения, а также создание и использование ссылок на методы. Подробно объясняются возможности использования лямбда-выражений для проведения математических вычислений, в том числе задач численных методов, таких как нахождение решения нелинейных уравнений и приближенное вычисление определенных интегралов. Издание ориентировано как на тех, кто не имеет опыта практического программирования на языке Java, так и на тех, кто хотел бы систематизировать и улучшить свои знания. В каждой теме приводится необходимый теоретический материал и код программ, что существенно ускоряет усваивание материала, а также способствует более квалифицированному подходу к программированию.

УДК 004.432.045:004.738.5Java (075.8)
ББК 32.973.2-018.1я73-1

Оглавление

Введение

Лямбда или лямбда-выражение представляет анонимную функцию или набор инструкций, которые можно выделить в отдельную переменную и затем многократно вызвать в различных местах программы или передавать в качестве параметра.

Передача метода в качестве аргумента другим методам с теоретической точки зрения представляет собой параметризацию поведения.

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. Функциональный интерфейс - это любой интерфейс, содержащий только один абстрактный метод. При этом дополнительно функциональный интерфейс может содержать один или несколько default-методов или статических методов.

Функциональные интерфейсы помечаются аннотацией `@FunctionalInterface`. Эта аннотация предназначена для того, чтобы сообщить компилятору, что данный интерфейс функциональный и должен содержать не более одного метода. Если же в интерфейсе с данной аннотацией более одного не реализованного (абстрактного) метода, компилятор не пропустит данный интерфейс, так как будет воспринимать его как ошибочный код. Интерфейсы и без указанной выше аннотации могут считаться функциональными и будут работать, а `@FunctionalInterface` является не более чем дополнительной страховкой.

Замечание.

Для компактности изложения в примерах не будем помечать интерфейсы как `@FunctionalInterface`.

Пример.

```
1  interface Operationable {
2      int calculate(int x, int y);
3  }
4
```

```
5  public class Lambda
6  {
7      public static void main(String[] args) {
8          Operationable operation;
9          operation = (x,y) -> x + y;
10         int result = operation.calculate(10, 20);
11         System.out.println(result);
12     }
13 }
```

Результат работы программы:

```
30
```

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.

По факту лямбда-выражения являются в некотором роде сокращенной формой внутренних анонимных классов, примеры которых неоднократно рассматривались ранее. В частности, предыдущий пример можно для анонимного класса переписать следующим образом.

Пример.

```
1  interface Operationable {
2      int calculate(int x, int y);
3  }
4
5  public class Lambda
6  {
7      public static void main(String[] args) {
8          Operationable op = new Operationable() {
9              //тело анонимного класса
10             public int calculate(int x, int y) {
11                 return x + y;
12             }
13         }; //конец анонимного класса
14         int z = op.calculate(20, 10);
15         System.out.println(z);
16     }
17 }
```

Результат работы программы такой же как в предыдущем случае.

Как видно из примеров выше эволюция анонимного класса в лямбда начинается с того, что опускается не только конструктор анонимного класса, но и имя метода интерфейса. Так как метод единственный в интерфейсе, то и его имя можно не упоминать. Если тело метода состоит из одного оператора, то и фигурные скобки также можно опустить.

Кроме того, типы параметров метода также *можно* опустить - предполагается, что они известны из сигнатуры единственного абстрактного метода.

Порядок объявления и использования лямбда-выражения

Для того чтобы объявить и использовать лямбда-выражение, основная программа разбивается на ряд этапов:

- определение ссылки на функциональный интерфейс:

```
Operationable operation;
```

- создание лямбда-выражения:

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

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.

Так, в методе интерфейса оба параметра представляют тип `int`, значит, в теле лямбда-выражения можно их сложить. Результат сложения также представляет тип `int`, объект которого возвращается методом интерфейса;

- использование лямбда-выражения в виде вызова метода интерфейса:

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

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

При этом для одного функционального интерфейса можно определить множество лямбда-выражений.

Пример.

```
interface Operationable {
    int calculate(int x, int y);
}

public class Lambda
{
    public static void main(String[] args) {
        Operationable operation;
        Operationable operation1 = (int x, int y)-> x + y;
        Operationable operation2 = (int x, int y)-> x - y;
        Operationable operation3 = (int x, int y)-> x * y;
        int result = operation3.calculate(10, 20);
        System.out.println(result);
    }
}
```

Результат работы программы:

```
200
```

Отложенное выполнение

Одним из ключевых моментов в использовании лямбд является отложенное выполнение (`deferred execution`). То есть в одном месте программы определяется лямбда-выражение и затем его можно 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 типу параметрам метода из функционального интерфейса. При написании самого лямбда-выражения тип параметров писать необязательно, хотя как указано в примере выше в принципе это можно сделать.

Если метод не принимает никаких параметров, то пишутся пустые скобки.

Пример.

```
1  interface Operationable {
2      int calculate();
3  }
4
5  public class Lambda
6  {
7      public static void main(String[] args) {
8          Operationable operation;
9          operation = () -> 100 / 25;
10         int result = operation.calculate();
11         System.out.println(result);
12     }
13 }
```

Результат выполнения программы:

```
4
```

Если метод принимает только один параметр, то скобки и вовсе можно опустить.

Пример.

```
1  interface Operationable {
2      int calculate(int n);
3  }
4
5  public class Lambda
6  {
7      public static void main(String[] args) {
8          Operationable operation;
```

```
        operation = n -> n * n;
        int result = operation.calculate(10);
        System.out.println(result);
    }
}
```

Результат выполнения программы:

```
100
```

Разберемся, что происходит «под капотом» при использовании лямбда-выражений. Кратко процесс можно описать следующим образом:

1. находится определение функционального интерфейса и единственного абстрактного метода в нем;
2. сверяется прототип этого абстрактного метода и левая часть лямбда-выражения, стоящая до стрелочки;
3. тип аргументов лямбда-выражения определяется из соответствия типу параметров метода из функционального интерфейса.

Терминальные лямбда-выражения

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

Пример.

```
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!");
    }
}
```

Результат работы программы:

```
Hello, Java!
```

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`.

Пример.

```
interface Operation{
    int calculate(int x, int y);
}

public class Lambda
{
    //свойства класса
    static int x = 10;
    static int y = 20;
    public static void main(String[] args) {
        Operation op = (int x, int y)-> {
           if(y == 0) return 0;
           else       return x/y;
        }; //конец лямбды
        System.out.println(op.calculate(20, 10));
        System.out.println(op.calculate(20, 0));
    }
}
```

Результат работы программы:

```
2
0
```

Лямбды и локальные переменные

Внутри тела всех лямбда-выражений, которые использовались ранее, можно было взаимодействовать только их аргументами. Но в лямбда-выражениях, как и в анонимных классах, могут применяться еще так

называемые свободные переменные (`free variables`), или переменные, которые не являются параметрами и объявлены во внешней области видимости. Такие лямбда-выражения называются лямбда-выражениями с захватом переменных (`capturing lambdas`).

Лямбда-выражения могут использовать переменные экземпляра и статические переменные класса, а также переменные 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.

Пример.

```
interface Operation{
    int calculate();
}

public class Lambda
{
    //свойства класса
    static int x = 10;
    static int y = 20;
    public static void main(String[] args) {
        Operation op = () -> {
            //тело лямбды
            x = 30;
            return x + y;
        }; //конец лямбды
        System.out.println(op.calculate());
        System.out.println(x);
    }
}
```

Результат выполнения программы:

```
50
30
```

Переменные `x` и `y` объявлены на уровне класса (являются полями класса), и в лямбда-выражении их можно получить и даже изменить. Так, в данном случае после выполнения выражения изменяется значение переменной `x`.

Теперь рассмотрим другой пример - локальные переменные на уровне метода.

Пример.

```
interface Operation {
    int calculate();
}

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());
    }
}
```

Результат выполнения программы:

```
100
```

Локальные переменные уровня метода также можно использовать в лямбдах, но изменять их значение нельзя. Если попробовать это сделать, то среда разработки укажет на ошибку, что такую переменную (которую попытались изменить) надо пометить с помощью ключевого слова `final`, то есть сделать константой (`final int n = 70;`). Однако это необязательно.

Более того, нельзя 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.

Поясним, откуда взялись подобные ограничения, налагаемые на локальные переменные. Существует различие во внутренней реализации переменных экземпляра класса и локальных переменных. Переменные экземпляра класса (объекта) хранятся в куче, в то время как локальные переменные располагаются в стеке. Переменные, которые хранятся в куче, могут быть использованы разными потоками выполнения. Локальные переменные метода располагаются в стеке и неявным образом ограничены

своим потоком выполнения. Если разрешить захват изменяемых локальных переменных, это приведет к открытию новых потоконебезопасных, а значит, нежелательных возможностей. Поэтому налагается ограничение: лямбда-выражения не могут модифицировать содержимое локальных переменных метода, в котором описаны.

Обобщенный (параметризованный) 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.

Пример.

```
interface Operation<T> {
    T calc(T x, T y);
}

public class Lambda
{
   public static void main(String[] args) {
      Operation<Integer> operation1 = (x, y)-> x + y;
      //это не перегрузка и не переопределение; это
      //объект с новым именем
      Operation<String> operation2 = (x, y) -> x + y;
      System.out.println(operation1.calc(20, 10));
      System.out.println(operation2.calc("20","10"));
   }
}
```

Результат работы программы

```
30
2010
```

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

Лямбды как параметры и результаты методов

Лямбды как параметры методов

Одним из преимуществ лямбд в Java является то, что их можно передавать в качестве параметров в методы. При этом, используемый однократно метод (функцию) не нужно описывать. Это делает код более понятным и прозрачным, поскольку не нужно тратить время на поиски в исходном коде, чтобы понять, какое действие или их набор выполняется.

Возможность передавать методы в качестве аргумента другим методам с теоретической точки зрения представляет собой параметризацию поведения. Возможность передачи кода при параметризации поведения позволяет адаптироваться к частым сменам требований заказчика. Например, заказчику требовалась программа, которая описывала бы его библиотеку и выбирала книги по жанру. Затем заказчику захотелось выбирать книгу по году издания, а завтра он возможно захочет выбрать книги по цвету или весу.

Таким образом, как видно из примера выборки (фильтрации) данных, возможность передачи лямбд в качестве параметров в методы позволяет «безболезненно», а главное неограниченно расширять функциональность проекта. Кроме того, использование лямбда-выражений позволяет создавать более гибкий и более удобный для повторного использования код.

В следующем примере находится сумма только тех элементов массива, которые отвечают определенному требованию, передаваемому как параметр метода. В данном случае, суммируются только четные числа.

Пример.

```
interface Expression {
    boolean isEqualEven (int n);
}

public class Lambda
{
    private static int sum(int[] a, Expression func)
    {
        int result = 0;
        for(int i : a) {
            if (func.isEqualEven(i))
                result += i;
        }
```

```
        return result;
    }

    public static void main(String[] args) {
        Expression condition = (n) -> n%2 == 0;
        int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        System.out.println(sum(nums, condition));
    }
}
```

Результат выполнения программы:

```
20
```

Функциональный интерфейс `Expression` определяет метод `isEqualEven()`, который возвращает `true`, если в отношении числа `n` действует какое-нибудь равенство.

В основном классе программы определяется метод `sum()`, который вычисляет сумму всех элементов массива, соответствующих некоторому условию. Само условие передается через параметр `func` типа `Expression`. Причем на момент написания метода `sum` можем не знать, какое именно условие будет использоваться. Само же условие определяется в виде лямбда-выражения (`Expression func = (n)-> n%2 == 0;`).

Таким образом, в данном случае все числа должны быть четными или остаток от их деления на `2` должен быть равен `0`. Затем это лямбда-выражение передается в вызов метода `sum`.

Замечание.

Впервые перед статическим методом `sum` стоит спецификатор `private`. Он ничего не меняет для доступа из метода `main()`, т.к. оба метода члены одного класса `Lambda`, но выглядит неожиданно.

При этом можно не определять переменную интерфейса, а сразу передать в метод лямбда-выражение. В следующем примере переопределим функциональный интерфейс `Expression`. Теперь он определяет метод `isMore(int n)`, который возвращает `true`, если в отношении числа `n` действует какое-нибудь равенство.

Пример.

```
interface Expression {
    boolean isMore(int n);
}
```

```
4
5   public class Lambda
6   {
7       private static int sum(int[] a, Expression func)
8       {
9           int result = 0;
10          for(int i : a) {
11              if (func.isMore(i))
12                  result += i;
13          }
14          return result;
15      }
16
17      public static void main(String[] args) {
18          int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
19          //сумма чисел, со знач. больше 5
20          int x = sum(nums, (n)-> n > 5);
21          System.out.println(x);
22      }
23  }
```

Результат работы программы:

```
30
```

Приведем пример использования лямбда-выражения в параметре метода `forEach()` некоторых коллекций.

Пример.

```
1   import java.util.*;
2   public class Lambda
3   {
4      public static void main(String[] args) {
5         ArrayList evens = new ArrayList();
6         evens.add(10);
7         evens.add("Строка");
8         evens.add(30);
9         evens.add(true);
10        evens.forEach((n)-> {System.out.print(n + " ");});
11     }
12  }
```

Результат работы программы:

```
10 Строка 30 true
```

Ссылки на метод как параметры методов

В настоящее время в Java можно в качестве параметра в метод передавать ссылку на другой метод. В принципе данный способ аналогичен передаче в метод лямбда-выражения.

Ссылки на методы можно рассматривать как вариант сокращенного написания лямбда-выражений, вызывающих только определенный метод. Если лямбда-выражение вызывает некий метод, то лучше сослаться на метод по названию, такая явная отсылка к названию метода повышает удобочитаемость кода.

Ссылка на метод передается в виде:

- `имяКласса::имяСтатическогоМетода` - если метод статический;
- `объектКласса::имяМетода` - если метод экземплярный.

Пример.

```
1  //функциональный интерфейс
2  interface Expression {
3      boolean action(int n);
4  }
5
6  //класс-справочник по методам
7  class Helper {
8      static boolean isEven(int n){
9          return n%2 == 0;
10     }
11     static boolean isPositive(int n){
12         return n > 0;
13     }
14 }
15
16 public class Lambda
17 {
18     private static int sum (int[] array, Expression f) {
19         int result = 0;
20         for(int i : array)
21         {
```

```
22              if (f.action(i))
23                  result += i;
24          }
25          return result;
26      }
27
28      public static void main(String[] args) {
29          int[] nums={ -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5};
30          System.out.println(sum(nums, Helper::isEven));
31
32          Expression expr = Helper::isPositive;
33          System.out.println(sum(nums, expr));
34      }
35  }
```

Результат работы программы:

```
0
15
```

Здесь также определен функциональный интерфейс `Expression`, который имеет один метод. Кроме того, определен класс `ExpressionHelper`, который содержит два статических метода. В принципе их можно было определить и в основном классе программы.

В основном классе `Lambda` метода `main()` определен метод `sum()`, который возвращает сумму элементов массива, соответствующих некоторому условию. Условие передается в виде объекта функционального интерфейса `Expression`.

В методе `main()` два раза вызываем метод `sum`, передавая в него один и тот же массив чисел, но разные условия. Первый вызов метода `sum`:

```
System.out.println(sum(nums, Helper::isEven));
```

На место второго параметра передается `Helper::isEven`, то есть ссылка на статический метод `isEven()` класса `Helper`. При этом методы, на которые идет ссылка, должны совпадать по параметрам и результату с методом функционального интерфейса.

При втором вызове метода `sum` отдельно создается ссылка `Expression` на функциональный объект, который затем передается в метод:

```
Expression expr = Helper::isPositive;
System.out.println(sum(nums, expr));
```

Приведем еще один пример в рамках использования ссылки на метод в качестве параметра метода `forEach()`.

Пример.

```
1  import java.util.*;
2  public class Lambda
3  {
4      public static void main(String[] args) {
5          ArrayList evens = new ArrayList();
6          evens.add(10);
7          evens.add("Строка");
8          evens.add(30);
9          evens.add(true);
10         evens.forEach(System.out::println);
11     }
12 }
```

Результат работы программы:

```
10
Строка
30
true
```

Таким образом использование ссылок на методы в качестве параметров аналогично использованию лямбда-выражений.

Если необходимо вызвать экземплярный метод, то в ссылке вместо имени класса применяется имя объекта этого класса:

Пример.

```
1  interface Expression {
2      boolean action(int n);
3  }
4
5  class Helper {
6      boolean isEven(int n){
7          return n%2 == 0;
8      }
9  }
10
11 public class Lambda
12 {
```

```
    private static int sum (int[] array, Expression f){
        int result = 0;
        for(int i : array) {
            if (f.action(i)) result += i;
        }
        return result;
    }

    public static void main(String[] args) {
        int[] nums = {0, 1, 2, 3, 4, 5};
        Helper help = new Helper();
        System.out.println(sum(nums, help::isEven));
    }
}
```

Результат работы программы:

```
6
```

Ссылки на конструкторы

Подобным образом можно использовать конструкторы: `ИмяКласса::new`.

Пример.

```
record User(String name) {}

interface UserBuilder{
    User create(String name);
}

public class Lambda
{
    public static void main(String[] args) {
        //ссылка на вызов конструктора
        UserBuilder ub = User::new;
        //вызов конструктора методом create(),
        //возвращающим тип User
        User user = ub.create("Tom");
        System.out.println(user.name());
    }
}
```

Результат работы программы:

При использовании конструкторов методы функциональных интерфейсов должны принимать тот же список параметров, что и конструкторы класса, и должны возвращать объект данного класса.

Лямбды как результат методов

Также метод в Java может возвращать лямбда-выражение. Рассмотрим следующий пример:

Пример.

```
interface Operation{
    int execute(int x, int y);
}

public class Lambda
{
    private static Operation action(int number){
        switch(number){
            case 1: return (x, y) -> x + y;
            case 2: return (x, y) -> x - y;
            case 3: return (x, y) -> x * y;
            default: return (x, y) -> 0;
        }
    }

    public static void main(String[] args) {
        Operation func = action(1);
        int a = func.execute(6, 5);
        System.out.println(a);

        int b = action(2).execute(8, 2);
        System.out.println(b);
    }
}
```

Результат работы программы:

В данном случае определен функциональный интерфейс `Operation`, в котором метод `execute()` принимает два значения типа `int` и возвращает значение типа `int`.

Метод `action()` принимает в качестве параметра число и в зависимости от его значения возвращает то или иное лямбда-выражение. Оно может представлять либо сложение, либо вычитание, либо умножение, либо просто возвращает `0`. Стоит учитывать, что формально возвращаемым типом метода `action()` является интерфейс `Operation`, а возвращаемое лямбда-выражение должно соответствовать этому интерфейсу.

В методе `main()` можно вызвать метод `action()`. Например, сначала получить его результат - лямбда-выражение, которое присваивается переменной `Operation`, а затем через метод `execute()` выполнить это лямбда-выражение:

```
Operation func = action(1);
int a = func.execute(6, 5);
System.out.println(a);        // 11
```

Либо можно сразу получить и тут же выполнить лямбда-выражение:

```
int b = action(2).execute(8, 2);
System.out.println(b);          // 6
```

Очевидно, также допустим синтаксис:

```
System.out.println(action(2).execute(8, 2));
```

Встроенные функциональные интерфейсы

В Java вместе с самой функциональностью лямбда-выражений также было добавлено некоторое количество встроенных функциональных интерфейсов, которые можно использовать в различных ситуациях и в различных `API`. Стандартные функциональные интерфейсы собраны в

пакете java.util.function и помечаются аннотацией @FunctionalInterface.

В частности, ряд далее рассматриваемых интерфейсов широко применяется в Stream API - прикладном интерфейсе для работы с данными (будет рассмотрен позже). Рассмотрим основные из этих интерфейсов:

- Predicate<T>;
- Consumer<T>;
- Function<T,R>;
- Supplier<T>;
- UnaryOperator<T>;
- BinaryOperator<T>.

Слово *предикат* (predicate) — выражение, использующее одну или более величину с результатом логического типа. Часто используется для обозначения функций, которые принимают значение в качестве аргумента и возвращают true или false. В Java определен функциональный интерфейс Predicate<T>:

```
public interface Predicate<T> {
    boolean test(T t);
}
```

Функциональным методом этого интерфейса является абстрактный метод test(), который проверяет соблюдение некоторого условия. Если оно соблюдается, то возвращается значение true. В качестве параметра лямбда-выражение принимает объект типа T. Этот интерфейс может пригодиться для представления булева выражения, в котором используется объект типа T.

Пример.

```
1  import java.util.function.Predicate;
2  public class Lambda
3  {
4     public static void main(String[] args) {
5        Predicate<Integer> isPositive = x -> x > 0;
6        System.out.println(isPositive.test(5));
7        System.out.println(isPositive.test(-7));
8     }
9  }
```

Результат работы программы:

```
true
false
```

Следующий функциональный интерфейс `BinaryOperator<T>`:

```
public interface BinaryOperator<T> {
    T apply(T t1, T t2);
}
```

Его метод `apply()` принимает в качестве параметра два объекта типа `T`, выполняет над ними бинарную операцию и возвращает ее результат также в виде объекта типа `T`.

Пример.

```
1  import java.util.function.BinaryOperator;
2  public class Lambda
3  {
4     public static void main(String[] args) {
5        BinaryOperator<Integer> multiply = (x, y) -> x*y;
6        System.out.println(multiply.apply(3, 5));
7        System.out.println(multiply.apply(10, -2));
8     }
9  }
```

Результат работы программы:

```
15
-20
```

Далее функциональный интерфейс `UnaryOperator<T>`:

```
public interface UnaryOperator<T> {
    T apply(T t);
}
```

Его метод `apply()` принимает в качестве параметра объект типа `T`, выполняет над ними операции и возвращает результат операций в виде объекта типа `T`:

Пример.

```
1  import java.util.function.UnaryOperator;
2  public class Lambda
3  {
4      public static void main(String[] args) {
5          UnaryOperator<Integer> square = x -> x * x;
6          System.out.println(square.apply(5));
7      }
8  }
```

Результат работы программы:

```
25
```

Функциональный интерфейс `Function<T,R>` имеет следующий синтаксис:

```
public interface Function<T, R> {
    R apply(T t);
}
```

Он представляет метод `apply()` перехода от объекта типа T к объекту типа R (приведение типов).

Пример.

```
1  import java.util.function.Function;
2  public class Lambda
3  {
4     public static void main(String[] args) {
5       Function<Integer,String> cast = x->String.valueOf(x)+
6                                       " штрафов";
7       System.out.println(cast.apply(7));
8     }
9  }
```

Результат выполнения программы:

```
7 штрафов
```

Еще один функциональный интерфейс `Consumer<T>` имеет формат:

```
public interface Consumer<T> {
    void accept(T t);
}
```

Его метод `accept()` выполняет некоторое действие над объектом типа T, при этом ничего не возвращая:

Пример.

```
1  import java.util.function.Consumer;
2  public class Lambda
3  {
4      public static void main(String[] args) {
5          Consumer<Integer> printer =
6              x-> System.out.printf("%d квитанций\n", x);
7          printer.accept(600);
8      }
9  }
```

Результат работы программы:

```
600 квитанций
```

Формат функционального интерфейса Supplier<T> имеет следующий вид:

```
public interface Supplier<T> {
    T get();
}
```

Его метод `get()` не принимает никаких аргументов, но должен возвращать объект типа `T`.

Пример.

```
1  import java.util.Scanner;
2  import java.util.function.Supplier;
3
4  record User(String name) {}
5
6  public class Lambda
7  {
8      public static void main(String[] args) {
```

```
        Supplier<User> userFactory = ()-> {
            //тело лямбды userFactory
            Scanner in = new Scanner(System.in);
            System.out.println("Введите имя: ");
            String name = in.nextLine();
            return new User(name);
        }; //конец лямбды
        User user1 = userFactory.get();
        User user2 = userFactory.get();
        System.out.println("Имя user1: " +
                                user1.name());
        System.out.println("Имя user2: " +
                                user2.name());
    }
}
```

Результат работы программы:

```
Введите имя:
Иван
Введите имя:
Сергей
Имя user1: Иван
Имя user2: Сергей
```

Встроенные функциональные интерфейсы для примитивных типов данных

Напомним, что типы данных Java делятся на ссылочные, например, `Byte`, `Integer`, `Object`, `List`, и примитивные (базовые) типы, например, `int`, `double`, `byte`, `char`. Параметры в описании рассмотренных ранее встроенных функциональных интерфейсов допускают использование только ссылочных типов.

Напомним также, что в Java существует механизм преобразования примитивных типов данных в соответствующие ссылочные типы. Этот механизм называется упаковкой или `boxing`. Обратный процесс, преобразование ссылочного типа данных в соответствующий примитивный тип называется распаковкой или `unboxing`. В Java существует также механизм автоупаковки (`autoboxing`), при котором для упрощения работы программистов упаковка и распаковка производятся автоматически.

Очевидно, что упакованные значения хранятся в куче. Следовательно, упакованные значения требуют большего количества памяти и дополнительных операций для извлечения обернутых простых значений.

Для повышения эффективности работы лямбд с примитивными (базовыми) типами в Java есть узкоспециализированные версии функциональных интерфейсов, позволяющие избежать выполнение операций автоупаковки. Некоторые из таких интерфейсов приведены в таблице 1.

Таблица 1. Распространенные функциональные интерфейсы

Функциональный интерфейс	**Варианты для примитивных типов данных**
`Predicate<T>`	`IntPredicate`, `LongPredicate`, `DoublePredicate`
`Consumer<T>`	`IntConsumer`, `LongConsumer`, `DoubleConsumer`
`Function<T, R>`	`IntFunction<R>`, `IntToDoubleFunction`, `IntToLongFunction`, `LongFunction<R>`, `LongToDoubleFunction`, `LongToIntFunction`, `DoubleFunction<R>`, `DoubleToIntFunction`, `DoubleToLongFunction`, `ToIntFunction<T>`, `ToDoubleFunction<T>`, `ToLongFunction<T>`
`Supplier<T>`	`BooleanSupplier`, `IntSupplier`, `LongSupplier`, `DoubleSupplier`
`UnaryOperator<T>`	`IntUnaryOperator`, `LongUnaryOperator`, `DoubleUnaryOperator`
`BinaryOperator<T>`	`IntBinaryOperator`, `LongBinaryOperator`, `DoubleBinaryOperator`

Примеры использования лямбда-выражений для проведения математических вычислений

Нахождение решения нелинейных уравнений

Обобщенная формулировка задания. Решить нелинейное уравнение вида $f(x)=0$ с параметром с заданной точностью ε, используя предлагаемые итерационные методы. Предварительно провести графическое отделение одного из корней, т.е. найти отрезок, на котором существует единственный корень. Требуемую точность и начальное приближение ввести с клавиатуры.

Требования:

- реализовать задание в виде многофайлового проекта с одним пакетом `lib`;
- значения начального приближения и точности вести с клавиатуры;
- оформить проект с использованием:
  - лямбда-выражения для функции $f(x)$ из индивидуального задания;
  - в случае реализации метода Ньютона также следует использовать лямбда-выражение для производной функции $f'(x)$.

Итерационные алгоритмы решения нелинейного уравнения. Использовать один из следующих итерационных алгоритмов нахождения решения на отрезке $[a,b]$:

1. Метод деления отрезка пополам (метод дихотомии):

Краткое описание метода для отрезка $[a,b]$*, содержащего единственный корень:* Пусть для определенности $f(a)<0$, $f(b)>0$. В качестве начального приближения корня $\tilde{x}$ принимается середина этого отрезка, т.е. $x_0=(a+b)/2$. Далее исследуем значение функции $f(x)$ на концах отрезков $[a\,;x_0\,]$ и $[\,x_0;b\,]$. Тот из них, на концах которого $f(x)$ принимает значения разных знаков, содержит искомый корень. Поэтому его принимаем в качестве нового отрезка, а вторую половину исходного отрезка $[a;\ b]$ отбрасываем. В качестве первой итерации нахождения корня принимаем середину нового отрезка и т. д. Если длина полученного отрезка становится меньше наперед заданной погрешности, т.е. $|\,b-a\,|<\varepsilon$, счет прекращается.

2. Метод простых итераций:

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{M}, \text{где } M > \max_{x\in[a,b]} |f'(x)|, n = \overline{0,\infty}, x_0 = a$$

3. Метод Ньютона

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, n = 0,1,\ldots,\ x_0 \in [a,b]$$

4. Модифицированный метод Ньютона

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_0)}, n = 0,1,\ldots,\ x_0 \in [a,b]$$

5. Метод секущих

$$x_{n+1} = x_n - \frac{(x_n - x_{n-1}) f(x_n)}{f(x_n) - f(x_{n-1})}, n = 0,1,\ldots x_0, x_1 \in [a,b]$$

Пример выполнения задания. Решить методом Ньютона с точностью ε нелинейное уравнение $3x - \cos x - 1 = 0$, предварительно отделив один корень.

Многофайловый проект должен иметь структуру, приведенную на рисунке (Рисунок 1).

Замечание.

При решении методом деления отрезка пополам файл `Derivative.java` *не нужен.*

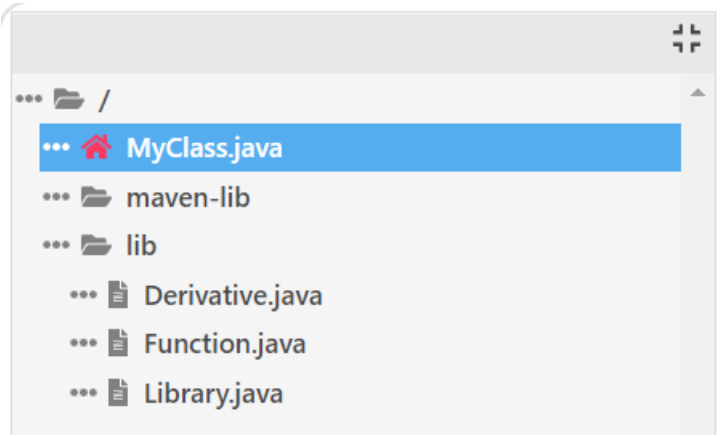

Рисунок 1 – Структура многофайлового проекта при решении нелинейного уравнения методом Ньютона

Пример проекта.

/MyClass.java

```java
import java.util.Scanner;
import lib.*;

public class MyClass
{
    public static void main(String[] args) {
        String exseptStr = "Неверный формат ввода числа";
        Scanner in = new Scanner(System.in);
        double x0, epsilon;

        while(true) {
            System.out.println("Введите начальное " +
                    "приближ. x0 и точность epsilon: ");
            try {
                x0 = Double.parseDouble(in.nextLine());
                epsilon = Double.parseDouble(in.nextLine());
            }
            catch (NumberFormatException e) {
                System.out.println(exseptStr);
                return;
            }
            if ((epsilon < 0.05) && (epsilon > 0)) break;
            System.out.println("Точность не верна." +
                               "Попробуйте еще раз!!!");
         }
         lib.Function f = (x) -> 3 *  x - Math.cos(x) - 1;
         lib.Derivative df = (x) -> 3 + Math.sin(x);
         System.out.println("Корень равен = " +
                  lib.Library.newtonMethod(x0, epsilon, f, df));
    }
}
```

/lib/Derivative.java

```java
package lib;

public interface Derivative {
    double lambda(double x);
}
```

/lib/Function.java

```java
package lib;

public interface Function {
    double lambda(double x);
}

```

/lib/Library.java

```
1  package lib;
2
3  public class Library {
4      public static double newtonMethod(double x0,
5                                        double epsilon,
6                                        Function f,
7                                        Derivative df) {
8      double prev = x0;
9      double next = prev -
10                   f.lambda(prev)/df.lambda(prev);
11     while (Math.abs(prev - next)>= epsilon)
12     {
13         prev = next;
14         next = prev -
15                f.lambda(prev)/df.lambda (prev);
16     }
17     return prev;
18     }
19 }
```

Результат работы программы:

```
Введите начальное приближ. x0 и точность epsilon:
1
0.005
Корень равен = 0.6071206581470717
```

Приближенное вычисление определенных интегралов

Обобщенная формулировка задания. Найти приближенное значение определенного интеграла $\int_a^b f(x)dx$ с заданной точностью ε, используя 2 из предложенных квадратурных формул.

Для достижения заданной точности использовать двойной пересчет: вычислить интеграл вначале для n разбиений отрезка, затем – для $2n$; сравнить полученные результаты: если $|I_n - I_{2n}| < \varepsilon$, то $I \approx I_{2n}$, в противоположном случае вычислить интеграл для $4n$ и т.д.

Подынтегральную функцию $f(x)$ и предлагаемую квадратурную формулу оформить в виде *лямбда-выражения*. Значения a, b, ε вести с клавиатуры.

Требования:

- реализовать задание в виде многофайлового проекта с тремя пакетами `lib`, `interfaces`, `library`;
- значения a, b, ε вести с клавиатуры;
- оформить проект с использованием:
  - лямбда-выражения для подынтегральной функции $f(x)$ из индивидуального задания;
  - лямбда-выражения для двух квадратурных формул из индивидуального задания.

Формулы для вычисления интегралов:

1. Составная формула трапеций:

$$\int_a^b f(x)dx \approx \frac{b-a}{2n}\left[f(a)+2f(a+h)+\ldots+2f(a+(n-1)h)+f(b)\right],$$

$$h=\frac{b-a}{n}.$$

2. Формула левых прямоугольников:

$$\int_a^b f(x)dx \approx h\sum_{i=0}^{n-1} f(x_i),\ h=\frac{b-a}{n},\ x_i=a+ih,\ i=\overline{o,n}.$$

3. Формула правых прямоугольников:

$$\int_a^b f(x)dx \approx h\sum_{i=1}^{n} f(x_i),\ h=\frac{b-a}{n},\ x_i=a+ih,\ i=\overline{o,n}.$$

4. Формула средних прямоугольников:

$$\int_a^b f(x)dx \approx h\sum_{i=0}^{n-1} f\left(x_i+\frac{h}{2}\right),\ h=\frac{b-a}{n},\ x_i=a+ih,\ i=\overline{o,n-1}.$$

5. Нижняя сумма Дарбу

$$\int_a^b f(x)\,dx \approx h\cdot\sum_{i=0}^{n-1} min\{f(x_i),f(x_{i+1})\},$$

$$h=\frac{b-a}{n},x_i=a+ih,i=\overline{0,n-1}.$$

6. Верхняя сумма Дарбу

$$\int_a^b f(x)\,dx \approx h\cdot\sum_{i=0}^{n-1} max\{f(x_i),f(x_{i+1})\},$$

$$h=\frac{b-a}{n},x_i=a+ih,i=\overline{0,n-1}.$$

Пример выполнения задания. Используя квадратурную формулу левых прямоугольников (2), вычислить приближенное значение интеграла $\int_a^b \sin x \cdot e^x dx$ с точностью ε.

Многофайловый проект должен иметь структуру, приведенную на рисунке (Рисунок **1** 3).

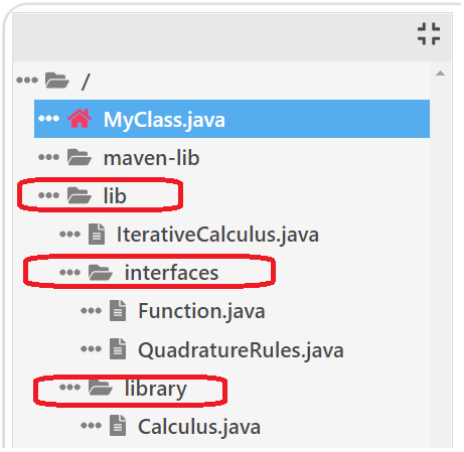

Рисунок 3 – Структура многофайлового проекта при вычислении интеграла с помощью правила левых прямоугольников (красным выделены пакеты)

Пример проекта.

/MyClass.java

```
1  import java.util.Scanner;
2  import lib.*;
3  import lib.interfaces.*;
4  import lib.library.*;
5
6  public class MyClass
7  {
8      public static void main(String[] args) {
9          String exseptStr = "Неверные формат ввода числа";
10         Scanner in = new Scanner(System.in);
11         double a = 0, b = 1, epsilon = 1;
12
13         while(true) {
14             System.out.println("Введите интервал интегрирования " +
15                                "[a, b] и точность epsilon: ");
16             try {
17                 //каждое число вводится в отдельной строке т.е. после Enter
18                 a = Double.parseDouble(in.nextLine());
19                 b = Double.parseDouble(in.nextLine());
20                 epsilon = Double.parseDouble(in.nextLine());
21             }
22             catch (NumberFormatException e) {
23                 System.out.println(exseptStr);
24             }
25             if ( (epsilon < 0.05) && (epsilon > 0) ) break;
26             System.out.println("Точность не верна. " +
```

```
                                    "Попробуйте еще раз!!!");
        }
        lib.interfaces.Function f = (x) -> Math.sin(x) * Math.exp(x);
        lib.interfaces.QuadratureRules sum = (begin, end, n, func) ->
                                    lib.library.Calculus.leftRectangles(begin, end, n, func);
        System.out.println("Интеграл равен = " +
                                lib.IterativeCalculus.sumIterator(a,b,epsilon,sum, f));
    }
}
```

/lib/IterativeCalculus.java

```
package lib;
import lib.interfaces.*;

public class IterativeCalculus {
    public static double sumIterator(double a,
                                     double b,
                                     double epsilon,
                                     QuadratureRules sum,
                                     Function f) {
        int n = 10;
        double current = sum.lambda(a, b, n, f);
        double next = sum.lambda(a, b, 2 * n, f);
        do {
            current = sum.lambda(a, b, n, f);
            next = sum.lambda(a, b, 2 * n, f);
            n = 2 * n;
        }
        while (Math.abs(current - next) > epsilon);
        return current;
    }
}
```

/lib/interfaces/Function.java

```
package lib.interfaces;

public interface Function {
    double lambda(double x);
}
```

/lib/interfaces/QuadratureRules.java

```
package lib.interfaces;

public interface QuadratureRules {
    double lambda(double a, double b, int n, Function f);
}
```

/lib/library/Calculus.java

```java
package lib.library;
import lib.interfaces.*;
import lib.*;

public class Calculus {

    public static double leftRectangles(double a,
                                        double b,
                                        int n, Function f) {
    double h = Math.abs(b - a) / n;
    double result = 0;
    for(double x = a; x < b; x += h) {
        result = result + f.lambda(x);
    }
    result = h * result;
    return result;
    }
}
```

Результат работы программы:

```
Введите интервал интегрирования [a, b] и точность epsilon:
1
2
0.005
Интеграл равен = 4.494596410233291
```

Литература

1. Руководство по языку программирования Java/ METANIT.COM - [Электронный ресурс], URL: https://metanit.com/java/tutorial/ (дата обращения: 08.06.22)
2. Блинов, И.Н. Java from EPAM / И.Н. Блинов, В.С. Романчик. - Минск: Четыре четверти, 2020. - 560 с.
3. Передача метода в качестве параметра в Java/ DelftStack - [Электронный ресурс], URL: https://www.delftstack.com/ru/howto/java/java-pass-method-as-parameter/ (дата обращения: 08.06.22)

Учебное издание

Кравчук Александр Степанович
Кравчук Анжелика Ивановна
Кремень Елена Васильевна

ЯЗЫК JAVA.

ЛЯМБДА-ВЫРАЖЕНИЯ

**Учебные материалы
для студентов специальности
1-31 03 08 «Математика
и информационные технологии
(по направлениям)»**

В авторской редакции

Ответственный за выпуск *Е. В. Кремень*

Подписано в печать 16.02.2023. Формат 60×84/16. Бумага офсетная.
Усл. печ. л. 2,32. Уч.- изд. л. 2,16. Тираж 50 экз. Заказ

Белорусский государственный университет.
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий № 1/270 от 03.04.2014.
Пр. Независимости 4, 220030, Минск.

Отпечатано с оригинал-макета заказчика
на копировально-множительной технике
механико-математического факультета
Белорусского государственного университета.
Пр. Независимости 4, 220030, Минск.