

## ЛАБОРАТОРНАЯ РАБОТА № 9

### «Наследование. Обработка исключительных ситуаций»

**Цель:** знакомство с иерархией классов исключений и получение навыков обработки ошибок.

#### **Учебные вопросы:**

1. Наследование;
2. Иерархия исключений;
3. Обработка исключений;
4. Оператор Throws;
5. Задания для самостоятельной работы;
6. Описание результата выполнения лабораторной работы.

# 1 Наследование

Для того чтобы разобраться в работе классов объектов **Java**, в том числе и исключений, следует коснуться фундаментального принципа объектно-ориентированного программирования – наследования – и пояснить что это такое.

Прежде всего, необходимо отметить, что все классы **Java** являются наследниками каких-то других классов, как показано в иерархии классов на рисунке 1.

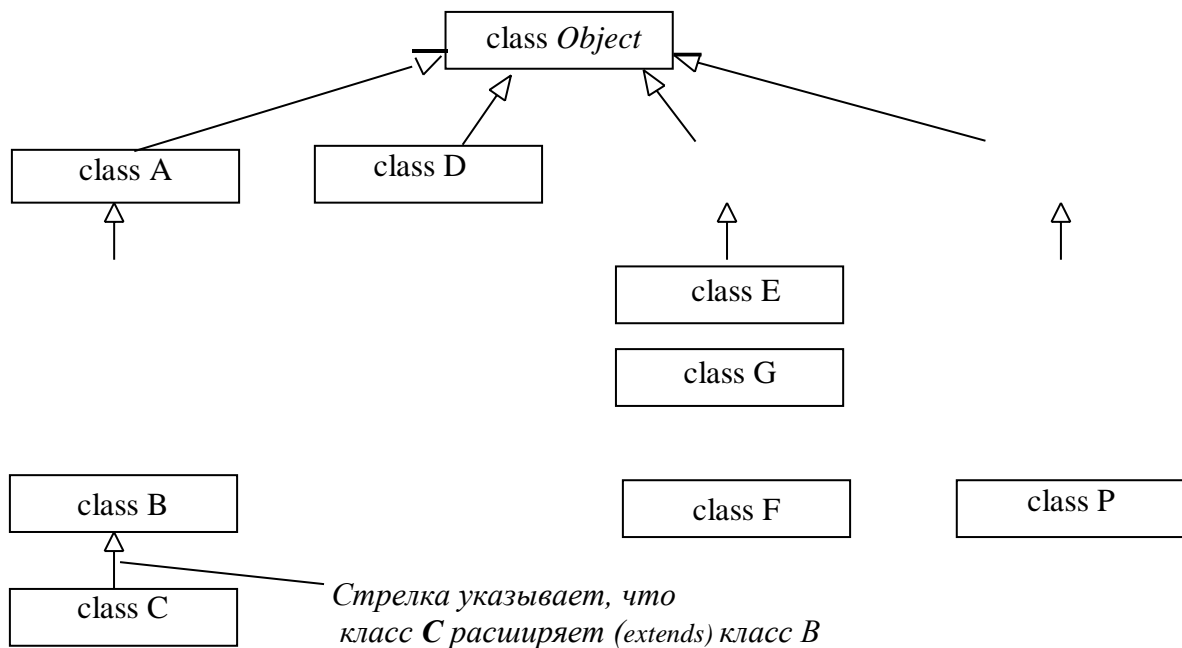


Рисунок 1

Наследование одного класса от другого можно записать так:

`class C extends B{... // класс B – родитель, класс C – потомок`

```
    содержимое класса
}
```

Класс-потомок (подкласс) наследует все методы класса- предка (суперкласса), а также расширяет (**extends**) родительский класс (суперкласс), добавляя ему новые методы и, соответственно, увеличивая его функциональность.

На вершине иерархии классов находится класс **Object**. Он явно или косвенно наследуется всеми классами. Если в сигнатуре (заголовке) какого-либо класса не указан другой класс, от которого он порожден, то по умолчанию считается, что он является наследником непосредственно класса **Object**. Все классы в Java наследники класса **Object**.

Каждый класс-потомок может иметь только один класс-предок. Так как всем классам-наследникам доступны методы их родителей, методы класса **Object** доступны всем другим классам. В качестве справки в табл. 1 приведены некоторые из методов класса **Object**, которые можно использовать в любой программе.

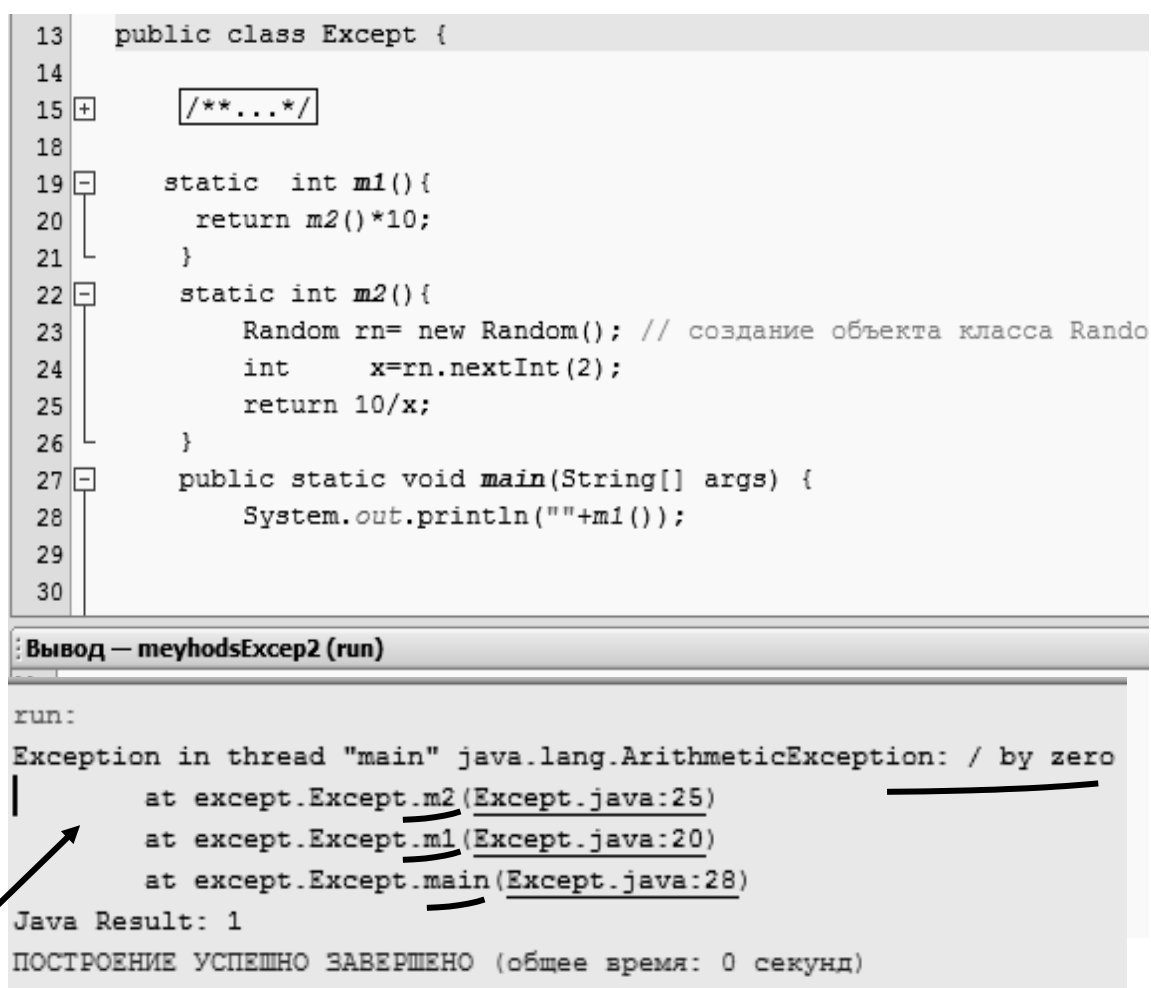
Таблица 1

<code>public String toString()</code>	Возвращает строковое представление объекта. По умолчанию возвращает строку, содержащую наименование класса, которому принадлежит текущий объект, символ @ и шестнадцатеричное представление хеш-кода объекта
<code>public final Class getClass()</code>	Возвращает объект типа Class, который представляет информацию о классе текущего объекта на этапе выполнения программы
<code>public int hashCode()</code>	Возвращает значение <i>хеш-кода</i> (hash code) текущего объекта
<code>protected Object clone() throws CloneNotSupportedException</code>	Возвращает клон текущего объекта. <i>Клон</i> – это новый объект, являющийся копией текущего

## 2 Иерархия исключений

Исключение – это ошибка, возникающая во время исполнения программы (но не в процессе компиляции). В JVM (java-машине) предусмотрена реакция на любую ошибку – автоматическое срабатывание обработчика исключений по умолчанию. В результате

этого программа завершает свою работу, а пользователь видит на экране сформированную трассу стека (**Stack Trace**), где указывается класс, соответствующий перехваченному исключению, место расположения ошибки и последовательность вызываемых методов, через которые эта ошибка передается («летит»). На рисунке 2 показано, что при использовании чисел, сгенерированных случайным образом, возможно возникновение ситуации деления на ноль (**/ by zero**).



```
13 public class Except {
14
15     /**...*/
16
17
18
19     static int m1(){
20         return m2()*10;
21     }
22     static int m2(){
23         Random rn= new Random(); // создание объекта класса Random
24         int x=rn.nextInt(2);
25         return 10/x;
26     }
27     public static void main(String[] args) {
28         System.out.println(""+m1());
29     }
30 }
```

Выход — meyhodsExcept2 (run)

run:  
Exception in thread "main" java.lang.ArithmeticException: / by zero  
at except.Except.m2(Except.java:25)  
at except.Except.m1(Except.java:20)  
at except.Except.main(Except.java:28)  
Java Result: 1  
ПОСТРОЕНИЕ УСПЕШНО ЗАВЕРШЕНО (общее время: 0 секунд)

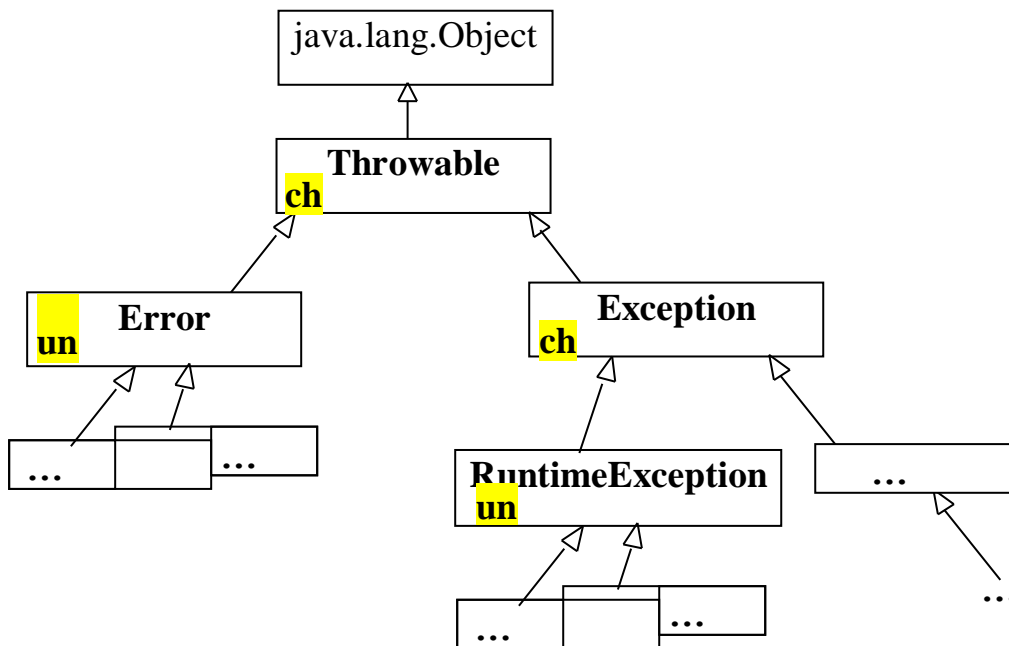
Трасса стека (Stack Trace) читается снизу вверх следующим образом:

метод **main** вызвал метод **m1()**, который вызвал метод **m2()**, где произошла исключительная ситуация **Exception** (ошибка) — деление на ноль **/by zero**, в результате чего был создан и перехвачен экземпляр класса исключений **ArithmeticException** пакета **java.lang**

Рисунок 2

Чтобы разбираться в исключительных ситуациях и уметь корректно реагировать на них в программе, следует ознакомиться с иерархией

наследования классов исключений. На рисунке 3 представлены четыре базовых класса исключений: **Throwable**, **Error**, **Exception**, **RuntimeException** – от них наследуются все остальные классы исключений **Java**.



где **ch** – checked; **un** – unchecked

Рисунок 3

На вершине иерархии исключений стоит класс **Throwable**, являющийся наследником класса **Object**. Каждый из типов исключений является подклассом класса **Throwable**. Два непосредственных наследника класса **Throwable** – **Error** и **Exception** делят иерархию подклассов исключений на две различные ветви.

Каждый из базовых классов исключений имеет определенный статус, который нельзя изменять – это **checked/unchecked** (проверяемый/непроверяемый). Все остальные наследники классов исключений имеют такой же статус, как и базовый класс, от которого они порождены.

Если исключение **checked** (**Throwable**, **Exception** или их потомки), то при написании программы компилятор будет выдавать ошибку (подчеркивать красной волнистой линией) и требовать от разработчика программного обеспечения самостоятельно (вручную) перехватить и обработать ошибку.

Если исключение **unchecked** (**Error**, **RuntimeException** или их потомки), то компилятор не проверяет, может ли быть порождена ошибка в коде, и разработчик сам принимает решение, как поступать в данной ситуации. В случае, представленном на рисунке 2, возникала **unchecked** ошибка, которая в программе не перехватывалась.

Следует отметить, что обработка исключительных ситуаций класса **IOException** и его наследников, обеспечивающих безопасность работы с файлами, является одним из важных достоинств языка.

На рисунке 4 изображено более полное дерево классов исключений, где показаны наиболее часто используемые классы стандартных ошибок Java, и с которыми придется сталкиваться при выполнении лабораторных работ.

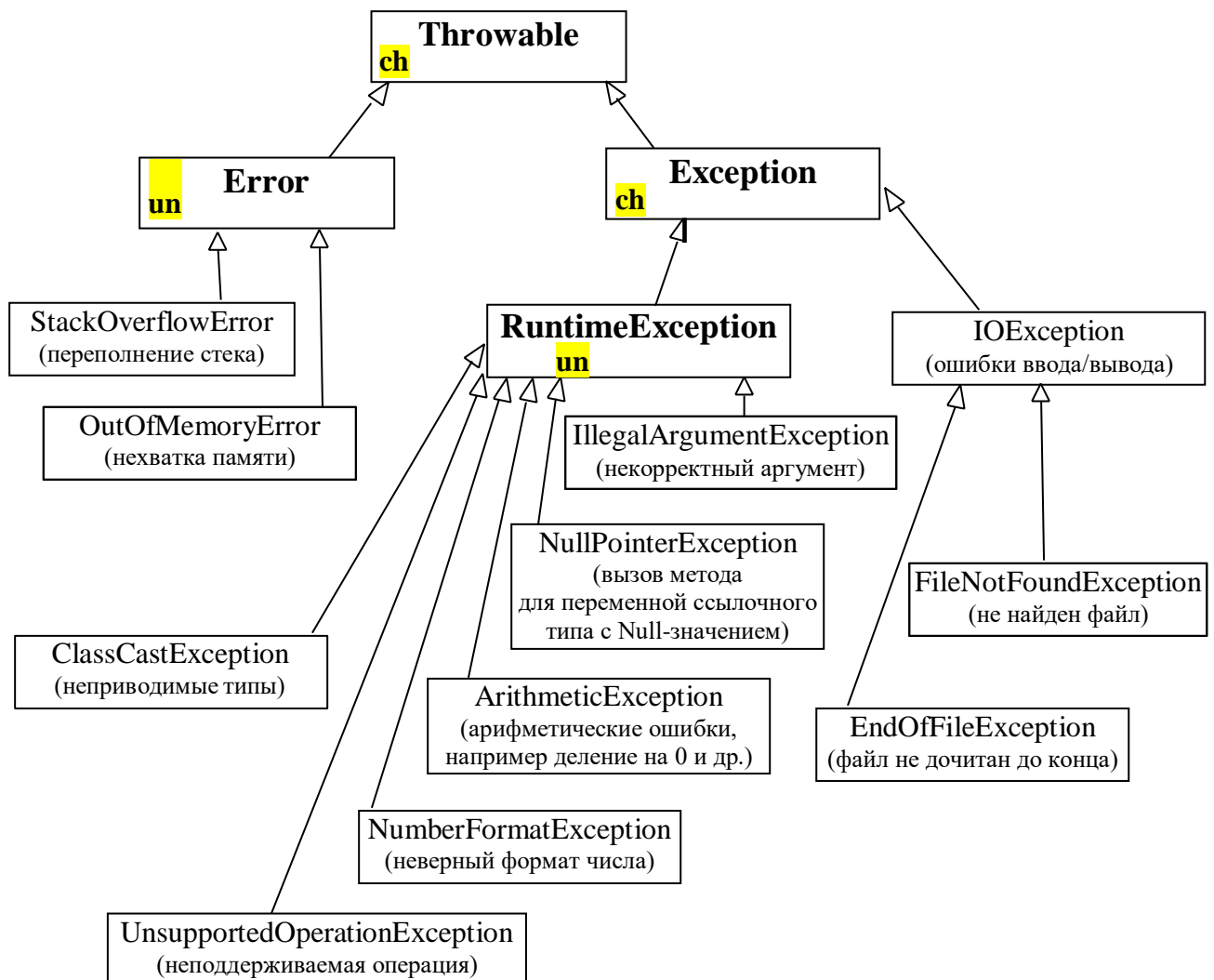


Рисунок 4

### 3 Обработка исключений

Для работы с экземплярами классов исключений используются пять ключевых слов: **try** – попытаться выполнить; **catch** – перехватить и обработать ошибку; **finally** – окончательно (финальный блок, выполняемый всегда); **throw** – генерация («бросание») исключения; **throws** – пометка метода, «бросающего» исключение.

Ниже приведена общая форма записи обработки исключений.

```
try {  
    // блок кода, вызывающего ошибку  
} catch (ТипИсключения1 e) {  
    // обработчик исключений типа ТипИсключения1  
} catch (ТипИсключения2 e) {  
    // обработчик исключений типа ТипИсключения2  
    throw(e) // возможно повторное возбуждение исключения  
}  
finally {  
}
```

Возможны следующие варианты использования блоков:

**try-catch** (или **try-catch-catch-catch...**);  
**try-catch-finally** (возможно: **try-catch-catch-catch-...-finally**);  
**try-finally**.

Сгенерировать необходимую ошибку можно, используя следующий синтаксис:

```
throw new Тип_исключения();
```

Тип исключения соответствует классу иерархии исключений стандартной библиотеки **Java** или созданного разработчиком и унаследованного от стандартного класса.

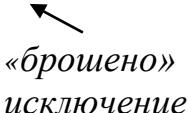
Несмотря на то, что самостоятельно создавать наследников, «бросать» исключения и перехватывать можно для любого класса

иерархии, не рекомендуется это делать для классов **Throwable** и **Error**. Автоматически экземпляры класса **Throwable** не создаются и не перехватываются. Обработка исключений класса **Error** и его наследников возлагается на **JVM**.

Разработчику рекомендуется работать с **checked** исключениями класса **Exception** и его наследниками и с **unchecked** исключениями класса **RuntimeException** и его наследниками.

Далее рассмотрим примеры обработки исключительных ситуаций.

**Пример 1.** Сгенерировано и перехвачено **RuntimeException**.

```
public static void main(String[] args) {  
    try {  
        System.out.println("0");  
        throw new RuntimeException("Непроверяемая ошибка");  
         создан экземпляр RuntimeException с сообщением  
«брошено»  
исключение»  
    } catch (RuntimeException e) { // исключение перехвачено  
        System.out.println("1 " + e); // исключение обработано  
    }  
    System.out.println("2");  
}
```

Предок может перехватывать исключения всех своих потомков.

**Пример 2.** Исключение перехвачено перехватчиком предка.

```
public static void main(String[] args) {  
    try {  
        System.out.println("0");  
        throw new RuntimeException("Непроверяемая ошибка");  
        System.out.println("1");  
    } catch (Exception e) {  
        System.out.println("2 " + e);  
    }  
    System.out.println("3");  
}
```



### **Пример 3.** Перехват исключения подходящим классом.

```
public static void main(String[] args) {  
    try {  
        System.out.println("0");  
        throw new RuntimeException("ошибка");  
    } catch (NullPointerException e) {  
        System.out.println("1" );  
    } catch (RuntimeException e) {  
        System.out.println("2" );  
    } catch (Exception e) {  
        System.out.println("3" );  
    }  
    System.out.println("4");  
}
```

### **Пример 4.** Перехват исключения подходящим классом.

```
public static void main(String[] args) {  
    try {  
        System.out.println("0");  
        throw new RuntimeException("ошибка");  
    } catch (NullPointerException e) {  
        System.out.println("1" );  
    } catch (Exception e) {  
        System.out.println("2" );  
    } catch (Error e) {  
        System.out.println("3" );  
    }  
    System.out.println("4");  
}
```

### **Пример 5.** Исключение не перехвачено.

```
public static void main(String[] args) {  
    try {  
        System.out.println("0");  
        throw new RuntimeException("ошибка");  
    } catch (NullPointerException e) {  
        System.out.println("1" );  
    }  
    System.out.println("2");  
}
```

**Пример 6.** Последовательность перехвата должна соответствовать иерархии классов исключений. Предок не должен перехватывать исключения раньше потомков. Указанный пример выдает ошибку компилятора. Программу запустить невозможно.

```
public static void main(String[] args) {  
    try {  
        System.out.println("0");  
        throw new NullPointerException("ошибка");  
    } catch (ArithmeticException e) {  
        System.out.println("1" );  
    } catch (Exception e) {  
        System.out.println("2" );  
    } catch (RuntimeException e) {  
        System.out.println("3" );  
    }  
    System.out.println("4");  
}
```

*← поменять местами обработчики*

**Пример 7.** Нельзя перехватить брошенное исключение с помощью чужого **catch**, даже если перехватчик подходит.

```
public static void main(String[] args) {  
    try {  
        System.out.println("0");  
        throw new NullPointerException("ошибка");  
    } catch (NullPointerException e) {  
        System.out.println("1" );  
        throw new ArithmeticException();  
    } catch (ArithmeticException e) {  
        System.out.println("2" );  
    }  
    System.out.println("3");  
}
```

*← для перехвата данного исключения необходимо создать новый обработчик*

Далее приведены примеры с использованием конструкции **try-finally**. Перехват брошенного исключения **catch** не производится. Секция **finally** выполняется всегда.

### Пример 8. Генерация исключения в методе.

```
public class Except1 {  
    public static int m(){  
        try {  
            System.out.println("0");  
            throw new RuntimeException();  
        } finally {  
            System.out.println("1");  
        }  
    }  
    public static void main(String[] args) {  
        System.out.println(m());  
    }  
}
```

### Пример 9. Генерация исключительной ситуации в методе и дополнительное использование оператора **return**.

```
public class Except2 {  
    public static int m(){  
        try {  
            System.out.println("0");  
            return 55;           // выход из метода  
        } finally {  
            System.out.println("1");  
        }  
    }  
    public static void main(String[] args) {  
        System.out.println(m());  
    }  
}
```

### Пример 10. Генерация исключительной ситуации в методе. Использование оператора **return** в секциях **try** и **finally**.

```
public class Except3 {  
    public static int m(){  
        try {  
            System.out.println("0");  
            return 15;  
        } finally {  
            System.out.println("1");  
            return 20;  
        }  
    }  
    public static void main(String[] args) {  
        System.out.println(m());  
    }  
}
```

### Пример 11.

```
public class Except4 {  
    public static void main(String[] args) {  
        try {  
            System.out.println("0");  
            throw new NullPointerException("ошибка");  
        } catch (NullPointerException e) {  
            System.out.println("1" );  
        } finally {  
            System.out.println("2" );  
        }  
        System.out.println("3");  
    }  
}
```

**Пример 12.** Исключение **IllegalArgumentException** – неверные аргументы.

```
public class Except5 {  
    public static void m(String str, double chislo){  
        if (str==null) {  
            throw new IllegalArgumentException("Строка введена неверно");  
        }  
        if (chislo>0.001) {  
            throw new IllegalArgumentException("Неверное число");  
        }  
    }  
  
    public static void main(String[] args) {  
        m(null,0.000001);  
    }  
}
```

**Пример 13.** Пример работы с аргументами метода **main**. На рисунке 5 представлена настройка проекта и задание входных значений аргументов.

```
public class Except6 {  
    public static void main(String[] args) {  
        try {  
            int l = args.length;  
            System.out.println("размер массива= " + l);  
        }  
    }  
}
```

```

    int h=10/l;
    args[l + 1] = "10";
} catch (ArithmeticException e) {
    System.out.println("Деление на ноль");
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Индекс не существует");
} }
}

```

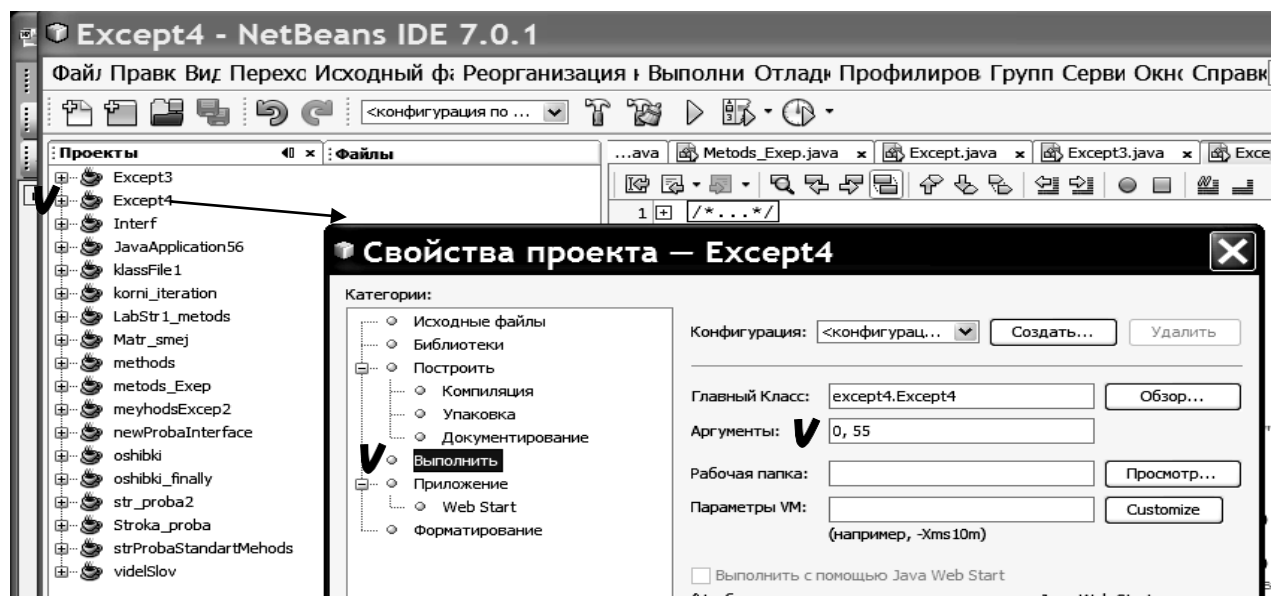


Рисунок 5

Через контекстное меню нужного проекта открыть диалоговое окно *Свойства* и установить нужные параметры аргументов метода *main*.

## 4 Оператор Throws

Если метод способен к порождению исключений, которые он не обрабатывает, он должен быть определен так, чтобы вызывающие методы могли сами предохранять от данного исключения. Для этого используется ключевое слово **throws** в сигнатуре метода.

Это необходимо для всех исключений, кроме исключений типа **Error** и **RuntimeException**, и, соответственно, для любых их подклассов.

**Пример 14.** Обработка исключения, порожденного одним методом *m()* в другом (в методе *main*).

```
public class Except7 {  
    public static void m(int x) throws ArithmeticException{  
        int h=10/x;  
    }  
    public static void main(String[] args) {  
        try {  
            int l = args.length;  
            System.out.println("размер массива= " + l);  
            m(l);  
        } catch (ArithmeticException e) {  
            System.out.println("Ошибка: Деление на ноль");  
        }  
    }  
}
```

## 5 ЗАДАНИЯ

**Задание 1.** Выполнить примеры 1-14 лабораторной работы, дав письменно объяснения (в комментариях к коду) последовательности выполняемых команд.

**Задание 2.** Выполнить все задания из таблицы 2:

– определить экспериментально, ошибки каких классов будут сгенерированы;

– создать обработчики исключительных ситуаций с использованием выявленных классов и всех секций конструкции обработчика с соответствующими сообщениями, позволяющими корректно выполнить программу.

Каждое задание выполнить в виде двух проектов: без использования собственных методов и с использованием методов для каждой подзадачи, которые могут генерировать исключительную ситуацию.

*Таблица 2*

### **Задание 1**

В программе, вычисляющей среднее значение среди положительных элементов одномерного массива (тип элементов `int`), вводимого с клавиатуры, могут возникать ошибки в следующих случаях:

- ввод строки вместо числа;
- несоответствие числового типа данных;
- положительные элементы отсутствуют.

### **Задание 2**

В программе, где требуется из матрицы вывести столбец с номером, заданным с клавиатуры, могут возникать ошибки в следующих случаях:

- ввод строки вместо числа;
- нет столбца с таким номером.

### **Задание 3**

В программе, вычисляющей сумму элементов типа `byte` одномерного массива, вводимого с клавиатуры, могут возникать ошибки в следующих случаях:

- ввод строки вместо числа;
- ввод или вычисление значения за границами диапазона типа.

## 6 ОПИСАНИЕ РЕЗУЛЬТАТА ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

В отчете по лабораторной работе должны быть представлены все примеры с описаниями вызываемых команд, а также 6 проектов заданий из задания 2.

Лабораторная работа принимается при наличии отчета всех выполненных заданий.

Структура отчета по лабораторной работе:

1. Титульный лист;
2. Цель работы;
3. Описание задачи;
4. Ход выполнения (содержит код программы);
5. Вывод;

Оформление:

- а) шрифт Times New Roman;
- б) размер шрифта 12 или 14;
- в) межстрочный интервал 1,5.

Отчет выполняется индивидуально и направляется по адресу электронной почты [proverkalab@yandex.ru](mailto:proverkalab@yandex.ru). В теле письма необходимо указать ФИО студента и номер группы.