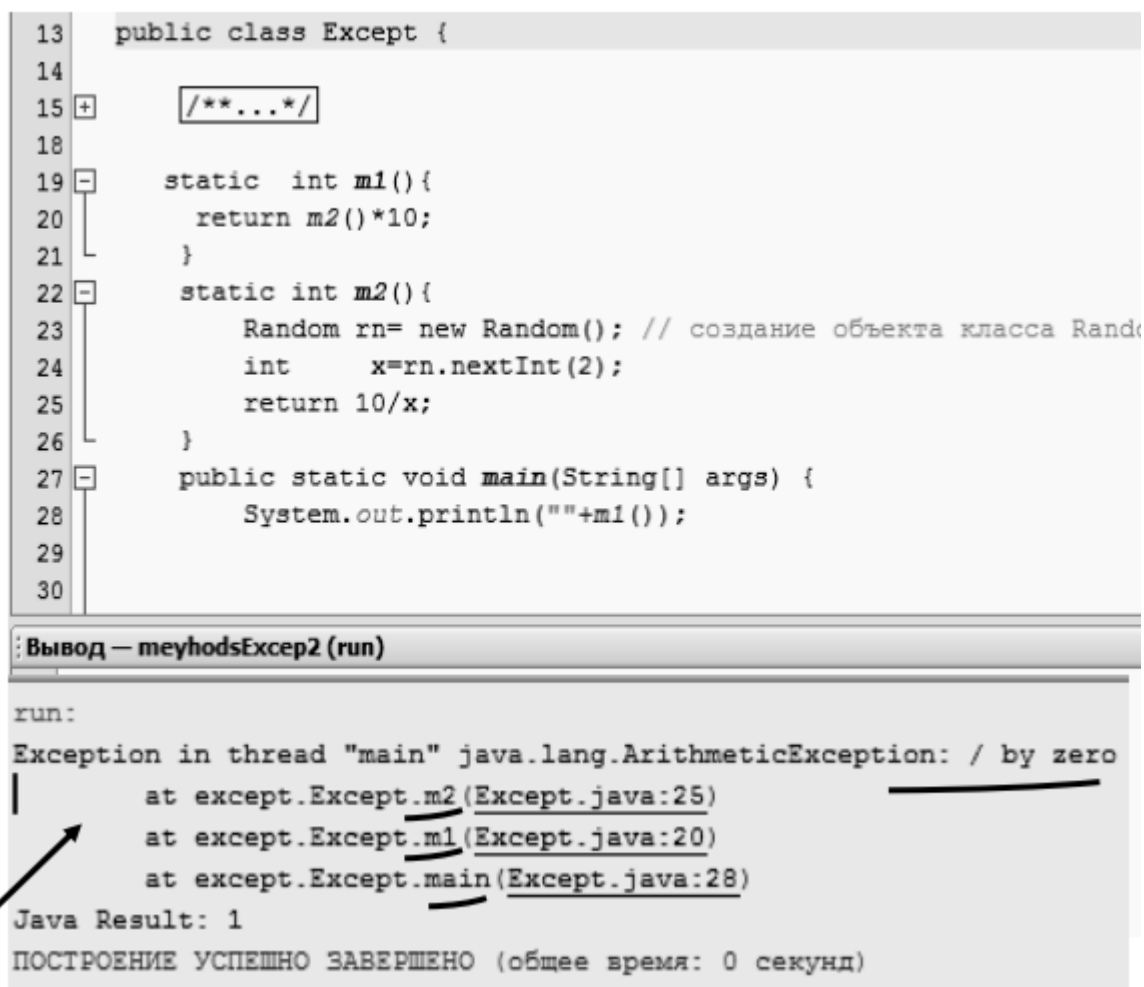


Иерархия исключений

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 рис. 4.2 показано, что при использовании чисел, сгенерированных случайным образом, возможно возникновение ситуации деления на ноль (/ **by zero**).



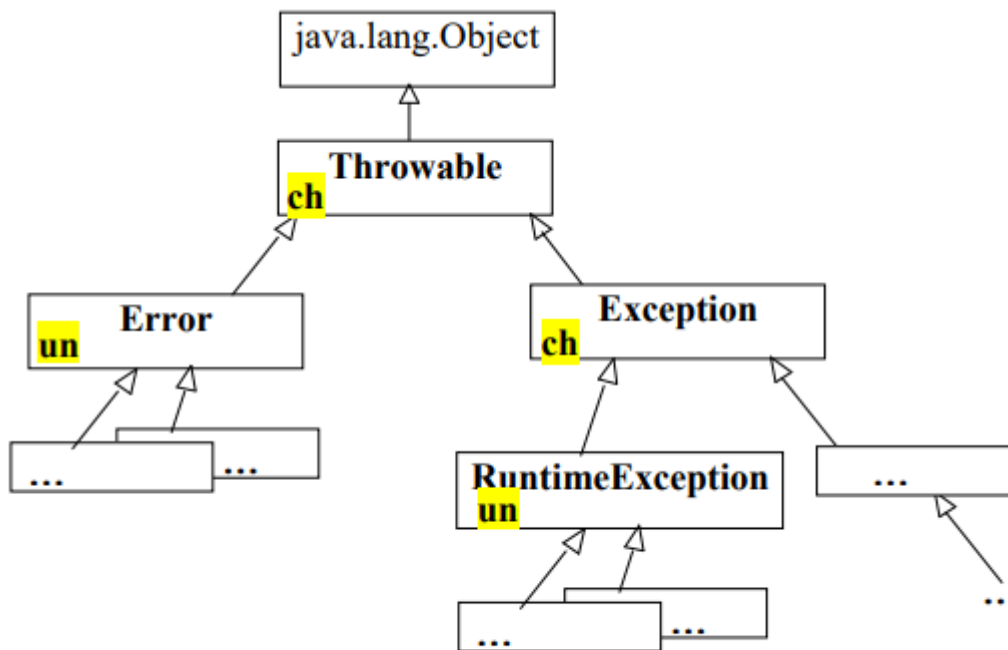
```
13 public class Except {
14
15     /**...*/
16
17
18
19     static int m1(){
20         return m2()*10;
21     }
22     static int m2(){
23         Random rn= new Random(); // создание объекта класса Random
24         int x= rn.nextInt(2);
25         return 10/x;
26     }
27     public static void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

Вывод — 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**

Чтобы разбираться в исключительных ситуациях и уметь корректно реагировать на них в программе, следует ознакомиться с иерархией наследования классов исключений. На рис. 4.3 представлены четыре базовых класса исключений: **Throwable**, **Error**, **Exception**, **RuntimeException** – от них наследуются все остальные классы исключений Java.

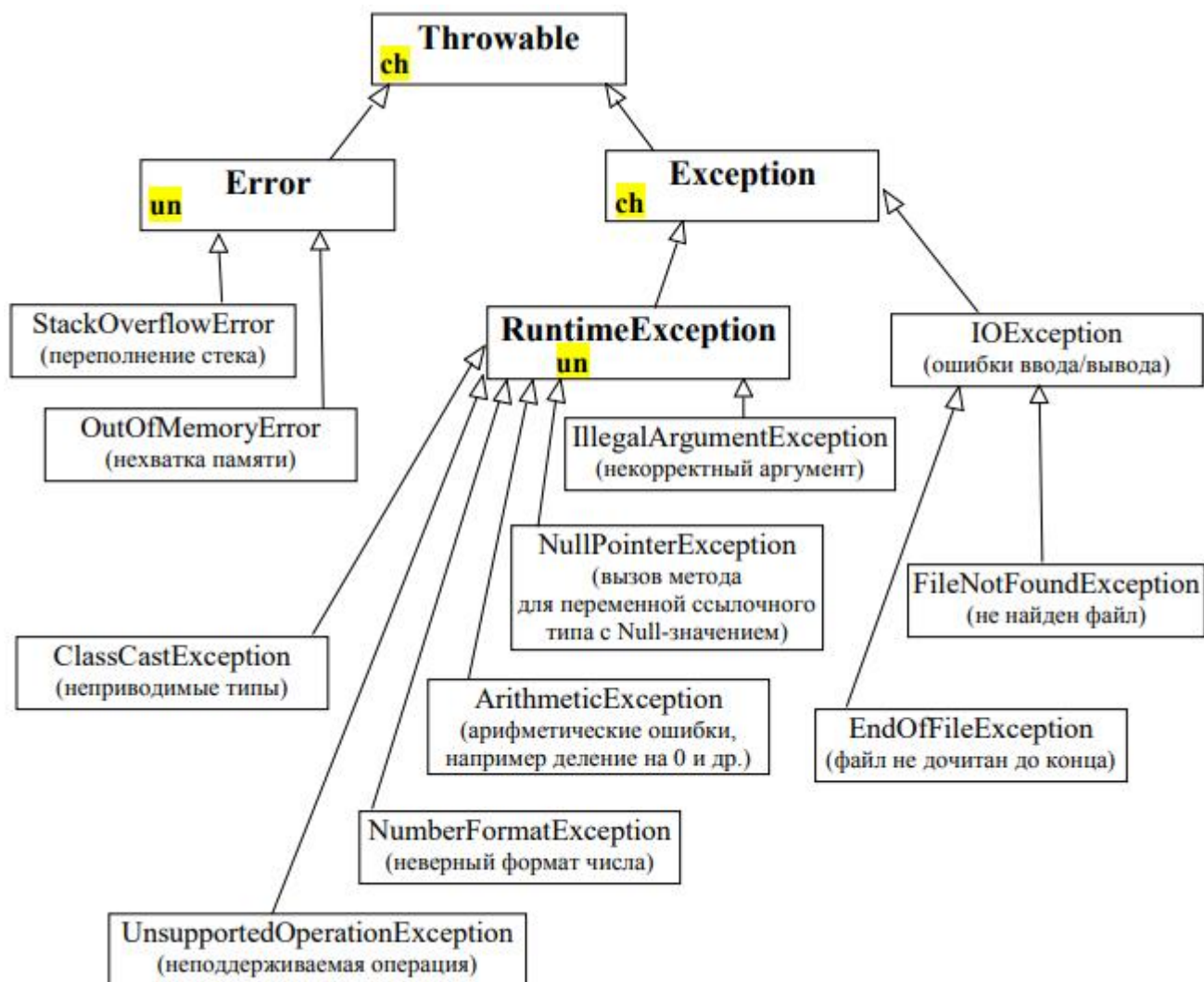


где **ch** – checked; **un** – unchecked

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 рис. 4.2, 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 рис. 4.4 изображено более полное дерево классов исключений, где показаны наиболее часто используемые классы стандартных ошибок Java, и с которыми придется сталкиваться при выполнении лабораторных работ.



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.

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.

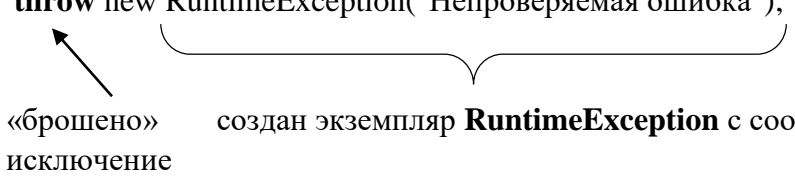
Несмотря на то, что самостоятельно создавать наследников, «бросать» исключения и перехватывать можно для любого класса иерархии, не рекомендуется это делать для классов **Throwable** и **Error**. Автоматически экземпляры класса **Throwable** не создаются и не перехватываются. Обработка исключений класса **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");  
    } catch (RuntimeException e) { // исключение перехвачено  
        System.out.println("1 " + e); // исключение обработано  
    } System.out.println("2"); }  
}
```



«брошено» исключение создан экземпляр **RuntimeException** с сообщением

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(); ← для перехвата данного исключения необходимо
} catch (ArithmeticException e) { создать новый обработчик
    System.out.println("2");
}
System.out.println("3");
}

```

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);
    }
}

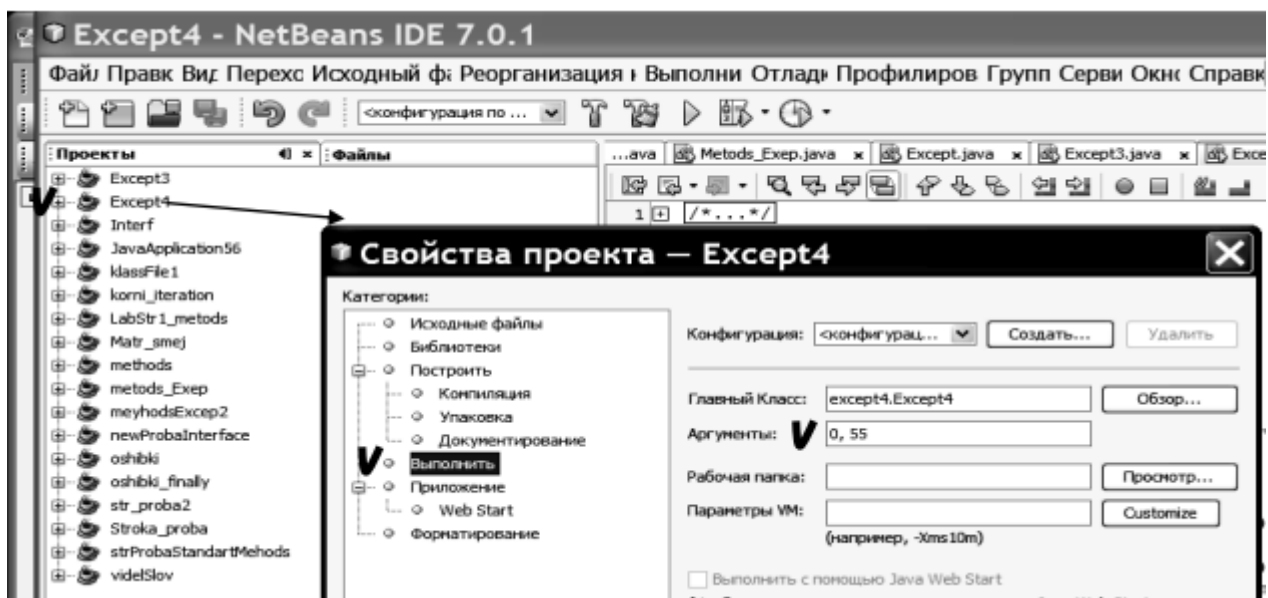
```

Пример 13. Пример работы с аргументами метода main. На рис. 2 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");
        }
    }
}

```



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

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 Excerpt7 {
    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");
        }
    }
}
```