Конспект лекций по Java. Занятие 13

В.Фесюнов

1. Обработка исключительных ситуаций (Exceptions)

Познакомимся с понятием *"исключительная ситуация"* (*исключение, прерывание, exception*). Применительно к данному термину используются обороты "исключение инициировано", "... выброшено", "... сгенерировано".

Исключительная ситуация может возникнуть при работе Java-программы в результате, например, деления на ноль или может быть инициирована программно внутри метода некоторого класса. Примером такого программно генерируемого исключения может служить FileNotFoundException, которое может быть выброшено методами классов ввода-вывода при попытке открыть несуществующий файл.

При возникновении исключения порождается объект, связанный с данным исключением. Соответственно, этот объект является объектом некоторого класса. Т.е. существует некоторое множество классов Java, которые вляются классами исключительных ситуаций. Так, FileNotFoundException — это класс. *Имя класса исключительной ситуации* и *имя исключения* употребляются как синонимы. Т.е., каждая исключительная ситуация имеет свое имя и это имя класса исключительной ситуации.

Объект, порождаемый при возникновении исключения, служит для хранения информации о возникшей исключительной ситуации (точка возникновения, описание и т.п.). Используя методы этого объекта можно, например, вывести на экран или в файл информацию о возникшем исключении.

Аппарат исключительных ситуаций (exceptions) является, в основном, средством обработки ошибок в программе. Он позволяет систематизировать и упростить обработку ошибок. Не следует применять этот аппарат для построения алгоритма

программы, т.е. базировать на нем логику программы. Он для этого не предназначен. Например, можно было бы запрограммировать генерацию исключения при нажатии на клавишу или на клик мыши. Но так поступать не следует, поскольку аппарат исключительных ситуаций для этого не достаточно эффективен.

Существуют два варианта генерации исключения — автоматическая генерация (примеры — NullPointerException, ArithmeticException, ClassCastException) и явная программная генерация.

Автоматическая генерация. Если Java-машина обнаруживает некоторую ошибку, например, деление на ноль или ошибку приведения типов, то она сама генерирует соответствующее исключение.

Программная генерация. Сгенерировать исключение можно явно при помощи операции **throw.** Это выглядит примерно так:

```
throw new IllegalArgumentException("Параметр k должен быть положительным числом");
```

Т.е. при генерации исключения мы порождаем объект класса нужной нам исключительной ситуации.

1.1. Поведение программы при возникновении исключения

В любом случае, и при автоматической и при программной генерации исключения, выполнение текущего потока команд программы прекращается.

- Если, например, поставить оператор сразу за оператором throw, то такой оператор никогда не выполнится.

Исключение можно перехватить. Для перехвата используется так называемый *try-catch* -блок. Но если не предпринять никаких действий по его перехвату, то выполнение текущего потока команд прекратится и будет выдано системное сообщение на консоль.

- В простейшем случае это приведет к прекращению выполнения самой программы, но не обязательно. Программа на Java может иметь несколько потоков выполнения (будем рассматривать это далее) и неперехваченное исключение приостановит только тот поток, в котором оно возникло. Так, любая диалоговая программа не

завершится при возникновении исключения, даже если его не перехватить, а только выдаст сообщение на консоль.

Итак, сгенерированное исключение прекращает выполнение текущего потока выполнения. Если в том методе, где возникло исключение, нет блока его перехвата, то метод прекращает свою работу. Если в методе, который вызвал данный метод, тоже нет блока перехвата, то и он завершается. И т.д., пока не будет обнаружен блок перехвата или не закончится цепочка вызванных методов.

1.2. Структура и использование блока перехвата исключений.

Для перехвата исключений используется синтаксическая конструкция, называемая *try-catch* -блок. Она имеет следующую структуру.

```
try {
   . . .
} catch ( Имя_класса_исключения ex ) {
   . . .
} catch  ...
```

Здесь внутри скобок после try находится блок, в котором мы собираемся ловить исключения. Т.е. исключение, возникшее в этом блоке, будет перехвачен данным try-catch-блоком (далее мы уточним, что не все возможные исключения перехватываются, а только указанные в конструкциях catch). В блоке *catch* , в его заголовке, указывается имя класса исключения, для которого предназначен этот блок. Блок будет срабатывать только при возникновении данного исключения или исключения, порожденного от данного (напомним, исключения — это классы, т.е. для них допустимо наследование классов).

Имя **ex** в заголовке catch-блока — это произвольное имя. Оно служит для именования объекта-исключения и с его помощью можно извлечь информацию о возникшем исключении.

Блоков catch в данном try-catch-блоке может быть несколько — каждый для своего исключения.

Пример

```
try {
    inputFile("data.txt");
    calculate();
} catch ( FileNotFoundException fe ) {
    System.out.println("Файл data.txt не найден");
} catch ( ArithmeticException aex ) {
    System.out.println("Деление на ноль");
    aex.printStackTrace(System.out);
}
```

В данном примере в try-блок заключены вызовы двух методов данного класса - inputFile() и calculate(). Если в них возникнет одна из двух исключительных ситуаций — FileNotFoundException или ArithmeticException, то она будут перехвачена и сработает соответствующий catch-блок. Если возникнет какое-то другое исключение, то оно не будет перехвачено и метод, в котором находится приведенный фрагмент (подразумеваем, что других try-catch-блоков в методе нет), будет прерван.

Рассмотрим еще один пример.

Пример

Пусть у нас описан следующий класс ExceptionExample1.

```
public class ExceptionExample1 {

  double f(double x) {
     try {
        return g(x, x-1)/x;
     } catch (ArithmeticException e ) {
        System.out.println("Деление на ноль");
        return 0;
     }
  }

  double g(double a, double b) {
      return Math.sin(a)/b;
  }
```

```
}
```

и где-то выполняется такой фрагмент:

```
ExceptionExample1 obj = new ExceptionExample1();
double y = obj.f(0);
```

Тогда при выполнении метода f(...) произойдет исключение при попытке деления на ноль, оно тут же будет перехвачено и в результате метод f(...) вернет ноль, а на консоль будет выдано сообщение "Деление на ноль".

Аналогичная ситуация произойдет и в случае

```
ExceptionExample1 obj = new ExceptionExample1();
double y = obj.f(1);
```

Однако здесь исключение произойдет в методе g(...). Он прекратит свою работу аварийно, но в методе f(...) возникшее в g(...) исключение будет перехвачено и произойдет то же самое, что и в первом случае.

Если бы f(...) не содержал try-catch-блока, то выполнение как первого, так и второго фрагмента закончилось бы аварийно. Если при этом сами фрагменты не находятся внутри try-catch-блока, то будет прерван и метод, в котором они находятся, и т.д. Если во всей цепочке вызовов методов не встретится соответствующий try-catch-блок, то прекратится и выполнение всего потока. Если этот поток единственный в программе, то прекратится выполнение всей программы.

Отметим еще один важный момент. В блоке catch для ArithmeticException стоит оператор

return 0;

Если попробовать его убрать, то транслятор выдаст ошибку. Действительно, в данной точке исключение обработано, программа продолжает свою нормальную работу, а, следовательно, метод f(...) должен вернуть какое-то значение. Т.е. try-catch-блок — это блок, который предназначен для восстановления нормальной работы программы после возникновения исключения.

Если же мы хотим просто выдать подходящее сообщение и фактически не перехватывать исключение (в данном примере это было бы естественней, чем перехват), то мы можем в блоке перехвата сгенерировать повторно данное исключение, т.е. вместо

retutn 0;

поставить

throw e;

Итак, мы не можем исключить из catch-блока оператор return, но можем заменить его на оператор throw. Это вызовет аварийное завершение метода f(...), что не требует возврата какого-либо значения. Такая замена с точки зрения транслятора Java будет корректной.

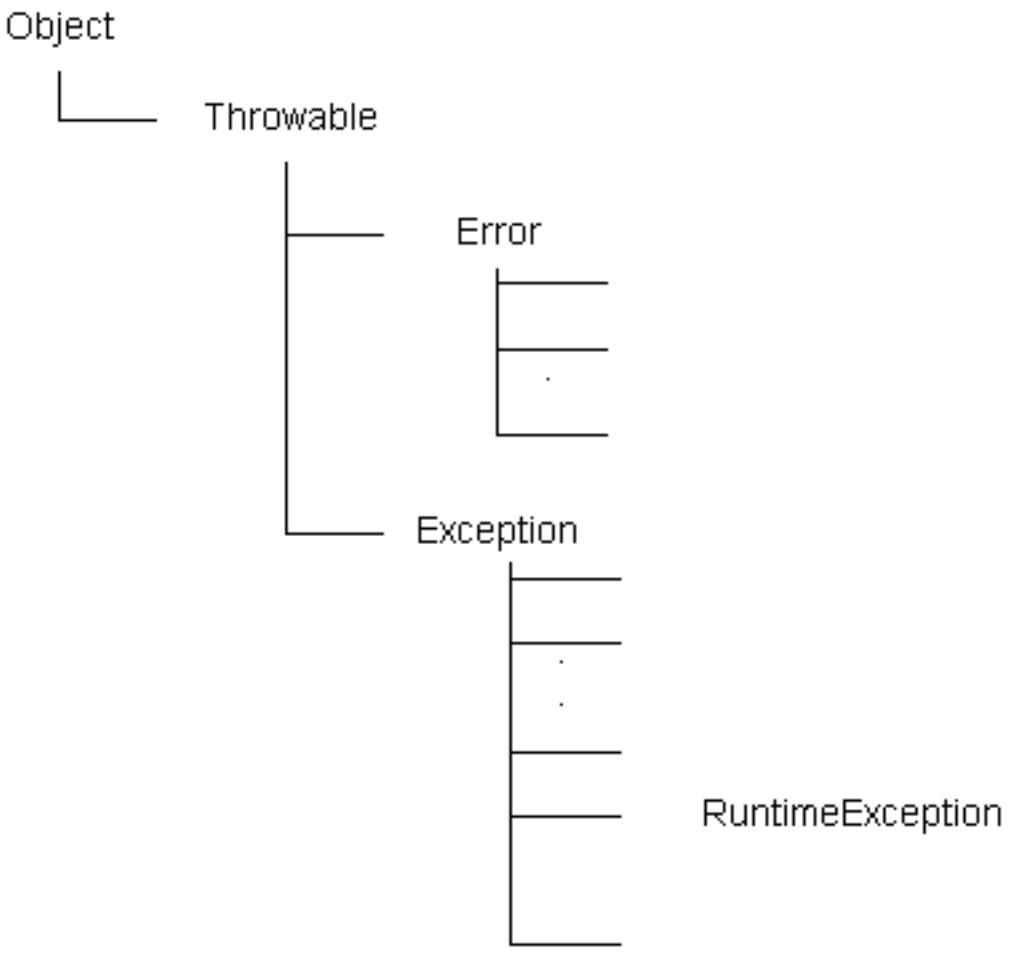
1.3. Классы исключительных ситуаций

Рассмотрим классы исключительных ситуаций. Это не произвольные классы Java. Они обладают своими особенностями и строятся по определенным правилам.

Правила построения классов исключительных ситуаций базируются на аппарате наследования и состоят в том, что классы исключительных ситуаций должны быть наследниками определенных классов стандартной библиотеки Java.

В стандартной библиотеке Java существует развитая иерархия классов исключительных ситуаций.

Рассмотрим эту иерархию.



Throwable — базовый класс для всех исключительных ситуаций.

Error — базовый класс для исключительных ситуаций, вызванных серьезными сбоями в работе виртуальной машины Java. Если возникла исключительная ситуация типа Error, то возможность продолжения работы программы сомнительна — нужно прекращать работу программы и, возможно, переинсталлировать Java. Перехватывать исключения типа Error не нужно.

Exception — это базовый класс для всех тех исключений, с которыми мы имеем дело в программах. Например, IOException порожден от Exception и может генерироваться различными методами библиотеки ввода/вывода. В свою очередь, для более точной спецификации исключений, от IOException порождены другие классы исключений,

такие, например, как FileNotFoundException.

Особое место занимает **RuntimeException** . Дело в том, что все Exception, кроме RuntimeException (т.е. все классы порожденные от Exception кроме тех, которые порождены от RuntimeException) обязаны быть перехвачены. Транслятор Java жестко контролирует это. Как именно - рассмотрим чуть позже. А пока разберемся для чего это нужно.

Различные исключения имеют различный характер. Исключение FileNotFoundException может возникнуть при попытке открытия несуществующего файла. В программе должна быть предусмотрена обработка этой ситуации.

Например, пользователь ввел неверное имя файла, что привело к возникновению исключения. Нехорошо, если в этом случае программа просто прекратит свою работу с выдачей непонятного рядовому пользователю сообщения. Пользователя нужно предупредить об этом "по человечески", а не выдачей системного сообщения об ошибке, и дать ему возможность ввести имя файла снова.

Поэтому Java заставляет разработчика перехватывать исключения подобного рода.

Другое дело исключение типа ArithmeticException. Оно может произойти при выполнении любой арифметической операции. Если бы пришлось его перехватывать, то вся программа состояла бы из одних try-catch-блоков, а это перебор. Поэтому исключения такого рода унаследованы от класса RuntimeException, что позволяет их не контролировать.

Это не означает, что их не следует вообще никогда перехватывать. Чаще всего известно, что в данном алгоритме может или не может возникнуть деление на ноль. Если оно может возникнуть, то это можно либо проверить явно, либо поставить try-catch-блок на ArithmeticException.

1.4. Механизм контроля перехвата исключений

Исключения, которые порождены от Exception, но не от RuntimeException, могут быть сгенерированы только явно операцией **throw** . Компилятор требует, чтобы в этом случае выполнялось одно из двух условий. Либо такой фрагмент должен находиться внутри try-catch-блока, который может перехватить данное исключение, либо в

заголовке того метода, где расположен данный фрагмент, должна стоять конструкция " *throws ИмяКлассаИсключения* ".

Например, мы рассматривали фрагмент программы, в котором фигурировал вызов метода

inputFile("data.txt");

По смыслу примера этот метод может генерировать FileNotFoundException. Поэтому его описание должно выглядеть примерно так:

```
public void inputFile( String fName ) throws FileNotFoundException {
 . . .
}
```

При этом в самом методе перехватывать это исключение не нужно.

Продолжим рассмотрение механизма контроля перехвата исключений.

В свою очередь, вызов метода, в описании которого стоит " throws ... ", тоже должен находиться либо внутри try-catch-блока, либо внутри метода с конструкцией " throws ... " в его заголовке. Таким образом, где-то в программе любое возможное исключение типа Exception обязано быть перехвачено.

Рассмотрим несколько рекомендаций по перехвату исключений и нюансов перехвата исключений, связанных с иерархией наследования классов исключений.

- **1.** Если некоторое исключение Ex1 имеет подкласс (порожденный класс) Ex2, то в catch-блок для Ex1 перехватит и исключение Ex2. Так в примере

```
try {
    inputFile("data.txt");
    calculate();
} catch ( FileNotFoundException fe ) {
    System.out.println("Файл data.txt не найден");
} catch ( ArithmeticException aex ) {
    System.out.println("Деление на ноль");
    aex.printStackTrace(System.out);
}
```

можно было вместо FileNotFoundException поставить IOException, эффект был бы тот же.

- **2.** Не рекомендуется строить несколько подряд идущих try-catch-блоков. Их следует, если это возможно, объединять в один блок. Тем самым обработка ошибок сосредотачивается в одном фрагменте программы, а не "размазывается" по всему ее тексту.
- **3.** Catch-блоки отрабатывают в порядке их появления в программе. Это нужно учитывать при их построении, в частности в связи с 1-м пунктом данных замечаний. Так во фрагменте

```
try {
  . . .
} catch ( IOException ex ) {
  . . .
} catch ( FileNotFoundException fex ) {
  . . .
}
```

второй catch-блок не отработает никогда, поскольку FileNotFoundException есть подкласс IOException. Если же их поставить в обратном порядке, то такая конструкция будет иметь смысл.

- **4.** Если необходимо перехватить любое исключение, нужно поставить catch-блок на Exception. Поскольку, согласно рассмотренной иерархии, Exception является предком всех исключений, то любое из них будет перехвачено данным catch-блоком. Однако нужно помнить, что такой блок должен быть последним в ряду catch-блоков.

2. Практическая работа

Разберем следующую программу.

```
import java.util.*;
import java.io.*;

public class InputTest {

  public static void main(String args[]) {
```

```
    if( args.length == 0 ) {
      System.out.println("Нужен параметр вызова: имя файла");
      return;
    }
    String thisLine;
    ArrayList list = new ArrayList();

    try {
        BufferedReader fin = new BufferedReader(new InputStreamReader(
                                new FileInputStream(args[0])));
        while ((thisLine = fin.readLine()) != null) {
            System.out.println("==Введена строка:"+thisLine);
            list.add(thisLine);
        }
    } catch (FileNotFoundException e) {
        System.out.println("failed to open file " + args[0]);
        System.out.println("Error: " + e);
        return;
    } catch (IOException e) {
        System.out.println("I/O error on file " + args[0]);
        System.out.println("Error: " + e);
        return;
    }
    Collections.sort(list);
    System.out.println("Отсортированный список строк:");
    Iterator iter = list.iterator();
    while( iter.hasNext() ) {
        String str = (String)iter.next();
        System.out.println(str);
    }
  }
}
```

Эта программа предназначена для ввода из файла и сортировки списка строк. Мы не изучали пока возможности ввода-вывода Java, поэтому рассмотрим соответствующие фрагменты на содержательном уровне, не вдаваясь в технические подробности.

Фрагмент

```
BufferedReader fin = new BufferedReader(new InputStreamReader(
                         new FileInputStream(args[0])));
```

служит для открытия файла, имя которого содержится в args[0], т.е. в первом параметре вызова программы. В частности, этот фрагмент содержит вызов конструктора FileInputStream(...). Если мы обратимся к документации по пакету java.io, то обнаружим, что соответствующий конструктор описан так:

public FileInputStream(String name) throws FileNotFoundException

Т.е. он содержит в описании фрагмент "throws FileNotFoundException". Если, в свою очередь, перейти по ссылке на описание FileNotFoundException, то мы обнаружим, что **FileNotFoundException** порожден от **Exception** , но не от **RuntimeException** . Это означает, что мы не можем включить данный фрагмент в программу без try-catch-блока, предназначенного для перехвата данного исключения.

В строке

while ((thisLine = fin.readLine()) != null) {

объект fin — это объект класса **BufferedReader** . Т.е. метод readLine() — это метод данного класса. Он читает и возвращает в качестве результата очередную строку файла.

Обратившись к документации по методу readLine() класса BufferedReader, мы обнаружим, что этот метод может выбрасывать исключение **IOException** , которое, как и FileNotFoundException требует перехвата.

Таким образом, мы должны организовать перехват как FileNotFoundException, так и IOException, что и сделано в нашей программе. Наш try-catch-блок содержит два catch-блока — один для FileNotFoundException, другой для IOException.

Следует обратить внимание, что оба фрагмента, требующие перехвата не разнесены по разным try-catch-блокам, а объединены в один фрагмент и заключены в единый try-catch-блок с двумя catch-блоками, каждый из которых предназначен для перехвата своего исключения.

Следует также обратить внимание на порядок следования этих catch-блоков - сначала

catch-блок для FileNotFoundException, а потом catch-блок для IOException. Это важно потому, что IOException вляется базовым классом для FileNotFoundException, и, если поставить их в обратном порядке, то блок для FileNotFoundException никогда не отработает — FileNotFoundException будет перехвачен блоком для IOException.

- Мы могли бы оставить один catch-блок для IOException, но тогда сообщение об ошибке было бы недостаточно информативным.

3. Блок finally

Это опциональный (необязательный) блок в конструкции try-catch. Если он присутствует, то он должен быть последним блоком в этой конструкции. Выглядит все это следующим образом:

```
try {
   . . .
} catch ( ... ) {
   . . .
} catch
   . . .
} finaly {
   . . .
}
```

В блок finally включают фрагмент программы, который должен выполняться в любом случае, вне зависимости от того, возникла ли исключительная ситуация или нет.

Возможность построения блока finally включена в язык для того, чтобы упростить освобождение захваченных ресурсов.

Рассмотрим предыдущий пример, улучшенный за счет применения блока finally.

```
import java.util.*;
import java.io.*;

public class InputTest {

  public static void main(String args[]) {
```

```
  if( args.length == 0 ) {
    System.out.println("Нужен параметр вызова: имя файла");
    return;
  }
  String thisLine;
  ArrayList list = new ArrayList();
  BufferedReader fin = null;

  try {
      fin = new BufferedReader(new InputStreamReader(
                                      new FileInputStream(args[0])));
      while ((thisLine = fin.readLine()) != null) {
          System.out.println("==Введена строка:"+thisLine);
          list.add(thisLine);
      }
      Collections.sort(list);
      System.out.println("Отсортированный список строк:");
      Iterator iter = list.iterator();
      while( iter.hasNext() ) {
          String str = (String)iter.next();
          System.out.println(str);
      }
  } catch (FileNotFoundException e) {
      System.out.println("Файл не найден: " + args[0]);
      System.out.println("Error: " + e);
  } catch (IOException e) {
      System.out.println("Ошибка ввода/вывода. Файл " + args[0]);
      System.out.println("Error: " + e);
  }  finally {
      if ( fin != null )
          try {
              fin.close();         // !!! Закрыть файл
          } catch ( IOException ex ) {
              System.out.println("Ошибка закрытия файла " + args[0]);
              System.out.println("Error: " + ex);
          }
      fin = null;
  }
}
```

```
}
```

В этом примере оператор fin.close() предназначен для закрытия файла. Он включен в блок finally, а это значит, что закрытие файла произойдет в любом случае. Оператор **if** в блоке finally нужен потому, что файл может и не открыться (опять же из-за исключения). В этом случае закрывать его не надо.

- Пример демонстрационный, поэтому в нем закрытие файла не имеет особого смысла, но в реальных программах желательно придерживаться правила, что все открытые файлы следует закрывать, как только потребность в них отпала.
- Данный пример является "шаблонным" в смысле обработки исключений ввода-вывода. Если нет на то особых причин, то при построении программ, выполняющих ввод-вывод, лучше использовать именно такой шаблон обработки исключений и закрытия файла после обработки.

4. Методы класса Throwable

Рассмотрим вопрос, какую информацию можно получить из исключения.

Основные возможности выдачи информации по исключению заложены в методах класса Throwable, который является базовым классом всей иерархии исключений. Он имеет ряд методов, но мы рассмотрим только наиболее важные из них.

public String toString()

Краткое сообщение о исключении.

public String getMessage()

Полное сообщение о исключении.

public void printStackTrace()

public void printStackTrace(PrintStream s)

public void printStackTrace(PrintWriter s)

Выдача в стандартный или указанный поток полной информации о точке возникновения исключения.

Пример

Программа Dlg4.java демонстрирует выдачу информации по исключению. В ней каждый третий клик на любой из 4-х кнопок вызывает генерацию исключения.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Dlg4 extends JFrame {

  int cnt = 0;

  Dlg4() {
     super("Знакомство с исключительными ситуациями");

     try  {
       UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
     }
     catch(Exception e) {
     }

    setSize(400, 200);
    Container c = getContentPane();
    c.setLayout(new GridLayout(2,2));
    JButton btnAry[] = new JButton[4];
    for ( int i = 0; i < 4; i++ ) {
        JPanel pn = new JPanel();
        btnAry[i] = new JButton(" Исключение "+(i+1));
        pn.add(btnAry[i]);
        c.add(pn);
    }
    btnAry[0].addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
           try {
               exceptTest(1);
           } catch ( Exception ex ) {
               ex.printStackTrace();
           }
```

```
        }
    });
    btnAry[1].addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
           try {
               exceptTest(2);
           } catch ( Exception ex ) {
               ex.printStackTrace();
           }
        }
    });
    btnAry[2].addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
           try {
               exceptTest(3);
           } catch ( Exception ex ) {
               ex.printStackTrace();
           }
        }
    });
    btnAry[3].addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
           try {
               exceptTest(4);
           } catch ( Exception ex ) {
               ex.printStackTrace();
           }
        }
    });
    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);

    setVisible(true);
  }

  void exceptTest(int btnNo) throws Exception {
```

```
      if ( ++cnt % 3 == 0 )
          throw new Exception("Сгененировано по кнопке N "+btnNo);
  }

  public static void main(String[] args) {
    Dlg4 d = new Dlg4();
  }
}
```

В данной программе исключение генерируется в методе exceptTest(...), который вызывается из соответствующего слушателя. Там же, в слушателе, организован перехват исключения и вывод информации на консоль по printStackTrace(). На консоль выдается следующее

```
java.lang.Exception: Сгененировано по кнопке N 1
    at Dlg4.exceptTest(Dlg4.java:76)
    at Dlg4$1.actionPerformed(Dlg4.java:31)
    at javax.swing.AbstractButton.fireActionPerformed(AbstractButton.java:1450)
    at и т.д.
```

Т.е. метод printStackTrace() выводит достаточно подробную информацию о точке возникновения исключения с указанием имени файла и номера строки файла.

5. Домашнее задание

Модифицировать программу InputTest.java следующим образом.

1). Она должна требовать не один, а два параметра вызова. Второй параметр — строка для поиска.

2). После сортировки и распечатки отсортированного списка программа должна искать в списке строку, заданную вторым параметром. Результат поиска нужно отпечатать.