# **ЛАБОРАТОРНАЯ РАБОТА №9 РАБОТА С ИСКЛЮЧЕНИЯМИ.**

**ЦЕЛЬ ЛАБОРАТОРНОЙ РАБОТЫ:**

Целью данной лабораторной работы являются получение практических навыков разработки программ, изучение синтаксиса языка Java, освоение основных конструкций языка Java (циклы, условия, создание переменных и массивов, создание методов, вызов методов), а также научиться осуществлять стандартный ввод/вывод данных.

Ключевые слова: try, catch, finally, throw, throws

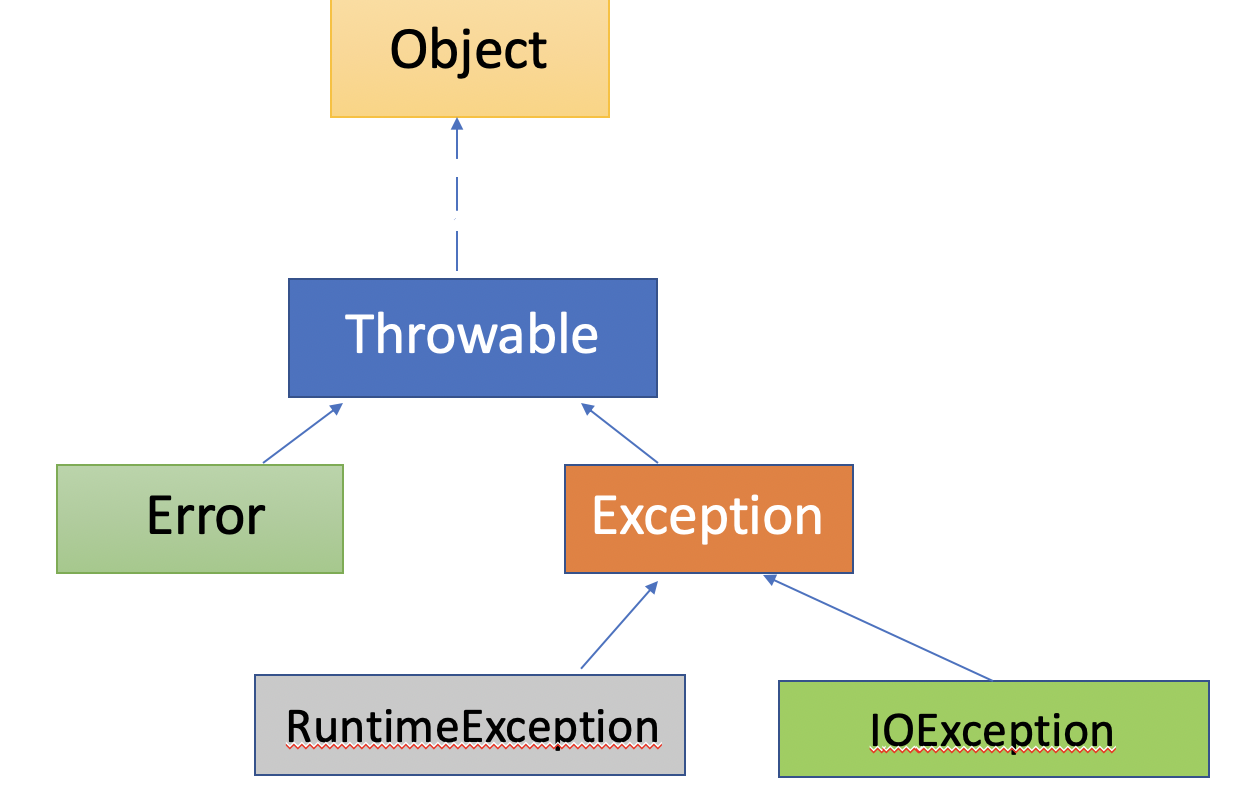
**ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ:**

Механизм исключительных ситуаций в Java поддерживается пятью ключевыми словами:

* **try**
* **catch**
* **finally**
* **throw**
* **throws**

В Java всего около 50 ключевых слов, и пять из них *связано* с *исключениями*: try, catch, finally, throw, throws. Из них catch, throw и throws применяются к экземплярам класса, причём работают они только с Throwable и его наследниками.

На рисунке 1 представлена иерархия классов исключений, используемая в Java.



**Рисунок 1 иерархия классов исключений**

Наиболее популярные исключений в Java представлены в таблице 1.

Таблица 1. Классы иcключений в Java

|  |  |  |
| --- | --- | --- |
| № пп | Класс исключения | Класс предок/тип |
|  | ArithmeticException | RuntimeException |
|  | NegativeArraySizeException | RuntimeException |
|  | ArrayIndexOutOfBoundsException | RuntimeException |
|  | NoSuchElementException | RuntimeException |
|  | ArrayStoreException | RuntimeException |
|  | NotSerializableException | Exception |
|  | AssertionError | Error |
|  | NullPointerException | RuntimeException |
|  | ClassCastException | RuntimeException |
|  | NumberFormatException | RuntimeException |
|  | ClassNotFoundException | Exception |
|  | OutOfMemoryError | Error |
|  | CloneNotSupportedException | Exception |
|  | SecurityException | RuntimeException |
|  | ConcurrentModificationException | RuntimeException |
|  | StackOverflowError | Error |
|  | EOFException | Exception |
|  | StringIndexOutOfBoundsException | RuntimeException |
|  | FileNotFoundException | Exception |
|  | ThreadDeath | Error |
|  | IllegalArgumentException | RuntimeException |
|  | UnsupportedEncodingException | Exception |
|  | InterruptedException | Exception |
|  | UnsupportedOperationException | RuntimeException |

То, что исключения являются объектами важно по двум причинам:

1. они образуют иерархию с корнем java.lang.Throwable (java.lang.Object — предок java.lang.Throwable, но Object —это не исключение!)
2. они могут иметь поля и методы

По первому пункту: catch — полиморфная конструкция, т.е. catch по типу класса родителя перехватывает исключения для экземпляров объектов как родительского класса, так или его наследников(т.е. экземпляры непосредственно самого родительского класса или любого его потомка).

Пример:

|  |
| --- |
| public class App {  public static void main(String[] args) {  try {  System.err.print(" 0");  if (true) {throw new RuntimeException();}  System.err.print(" 1");  } catch (Exception e) { // catch по Exception ПЕРЕХВАТЫВАЕТ RuntimeException  System.err.print(" 2");  }  System.err.println(" 3");  }// end main  } |

Результат работы программы:

>> 0 2 3

**Упражнения (Основы Try-Catch- Finally)**

**Задание № 1**

Выполните следующую программу и посмотрите, что происходит:

|  |
| --- |
| public class Exception1 {  public void exceptionDemo() {  System.out.println( 2 / 0 );  }  } |

**Листинг 1**

Необходимо инстанцировать класс и выполнить exceptionDemo (). Программа даст сбой, и вы получите следующее сообщение

java.lang.ArithmeticException: / by zero at Exception1.exceptionDemo(Exception1.java:12)

Это говорит нам о том, что программа пытается выполнить деление на ноль, который он не в состоянии выполнить.

Интересный эксперимент

Замените 2/0 на 2,0 / 0,0 и повторно вызовите метод. Что произойдет?

Теперь измените код в Exception1 чтобы включить try-catch block следующим образом:

|  |
| --- |
| public class Exception1 {  public void exceptionDemo() {  try{  System.out.println( 2 / 0 );  } catch ( ArithmeticException e ) {  System.out.println("Attempted division by zero");  }  }  } |

**Листинг 2**

Запустите программу и обратите внимание на новое поведение.

**Задание № 2**

Измените код в листинге 1 на следующий:

|  |
| --- |
| public class Exception2 {  public void exceptionDemo() {  Scanner myScanner = new Scanner( System.in );  System.out.print( "Enter an integer ");  String intString = myScanner.next();  int i = Integer.parseInt(intString); System.out.println( 2 / i );  }  } |

**Листинг 3**

Запустите эту программу со следующими выводами: Qwerty 0 1.2 1

Посмотрите на вывод. Какие исключения выбрасываются?

Измените код, добавив блоки try – catch, чтобы иметь дело с определяемыми исключениями**.**

**Задание № 3**

С помощью перехватывания исключений можно оказывать влияние на поведение программы. В вашем решении в предыдущем упражнении вы можете добавить новый пункт - catch в начале списка пунктов catch. Выполите это, чтобы поймать общее исключение класса Exception. Перезапустите программу с приведенными выше данными и обратите внимание на ее поведение. Объясните новое поведение программы

**Задание № 4**

И наконец добавьте блок finally к решению Задания №2. Повторно запустите программу, чтобы проверить ее поведение. Объясните новое поведение программы

**Генерация собственных исключений**

На предыдущем шаге при выполнении заданий мы рассмотрели, как Java работает с предопределенными исключениями, теперь перейдем к тому, как генерируется исключение.

Все исключения в рассмотренных ранее примерах и заданиях были определены заранее. В этом разделе лабораторной работы вы будете создавать и пробрасывать свои собственные исключения (exceptions).

**Задание № 5**

Самый простой способ генерации исключения показан в следующем примере кода:

|  |
| --- |
| public class ThrowsDemo {  public void getDetails(String key) {  if(key == null) {  throw new NullPointerException( "null key in getDetails" );  }  // do something with the key  }  } |

**Листинг 4**

Когда определяется условие ошибки, то мы выбрасываем исключение с определенным именем. Сообщение может быть связано с исключением. Откомпилируйте этот класс, создайте его экземпляр и выполните метод getDetails() с нулем в качестве значения параметра.

Вывод может быть следующим:

java.lang.NullPointerException: null key in getDetails at ThrowsDemo.getDetails(ThrowsDemo.java:13)

Добавьте блок try-catch таким образом, чтобы перехватить исключение и рассмотреть его внутри метода.

Подумайте, является ли этот способ подходящим, чтобы иметь дело с этим исключением? Объясните поведение программы.

Причиной ошибки, может является, например неправильное значение для параметра. Может было бы лучше, если бы метод вызывал getDetails() и там решалась бы эта проблема.

Обратите внимание на следующее:

|  |
| --- |
| public class ThrowsDemo {  public void printMessage(String key) {  String message = getDetails(key); System.out.println( message );  }  public String getDetails(String key) {  if(key == null) {  throw new NullPointerException( “null key in getDetails” );  }  return “data for” + key; }  } |

**Листинг 5**

Откомпилируйте и запустите эту программу с правильным значением для ключа и с нулем в качестве значения. При выполнении с нулевым значением вы увидите некоторый вывод. Обобщите все вышесказанное и выполните эту программу с правильным значением для ключа и с нулем в ключе.

java.lang.NullPointerException: null key in getDetails

at ThrowsDemo.getDetails(ThrowsDemo.java:21)  
 at ThrowsDemo.printMessage(ThrowsDemo.java:13)

Теперь добавьте блоки try-catch, чтобы использовать для вывода сообщений метод printMessage(), таким образом, чтобы исключения обрабатывались и программа не “ломалась”.

**Задание № 6**

Теперь мы расширим наш пример для демонстрации прохождения исключения через цепочку вызовов. Создайте следующий класс и попытатайтесь его скомпилировать:

|  |
| --- |
| public class ThrowsDemo {  public void getKey() {  Scanner myScanner = new Scanner( System.in );  String key = myScanner.next();  printDetails( key );  }  public void printDetails(String key) {  try { String message = getDetails(key);  System.out.println( message );  }catch ( Exception e){  throw e;  }  }  private String getDetails(String key) {  if(key == "") {  throw new Exception( "Key set to empty string" );  }  return "data for " + key; }  } |

**Листинг 6**

При попытке компиляции вы получите следующий синтаксис ошибки:

Unreported java.lang.Exception исключение;

В результате успешного пробрасывания исключение должен быть поймано или объявлено. Причиной полученной ошибки является выражение ***throw е.***

В данном случае метод printDetails () решил, что он не может иметь дело с исключением и проходит все дерево его вызовов. Поскольку метод GetKey() не имеет блока try-catch для обработки исключений, то Java становится перед выбором, что в таком случае делать.

Проблему можно решить несколькими способами:

1. Добавьте соответствующие try-catch блоки таким образом, чтобы в конечном итоге один из них обрабатывал исключение;
2. Удалите блоки try-catch для всех методов, кроме одного, который обрабатывает исключение. Добавьте throws, котрый бросает исключение методу, который проходит исключение без обработки.

**Задание № 7**

Измените код программы, чтобы она работала корректно.

Следующий фрагмент кода программы демонстрирует случай, когда никакие методы не будут брать на себя ответственность за обработку ошибок, и программа в конечном итоге выходит “сломается “.

|  |
| --- |
| public class ThrowsDemo {  public void getKey() throws Exception {  Scanner myScanner = new Scanner( System.in );  System.out.print("Enter Key ");  String key = myScanner.nextLine();  printDetails( key );  }  public void printDetails(String key) throws Exception {  String message = getDetails(key);  System.out.println( message ); }  private String getDetails(String key) throws Exception {  if(key.equals("")) {  throw new Exception( "Key set to empty string" );  }  return "data for " + key;  }  } |

**Листинг 7**

**Задание №8**

Измените этот код следующим образом:

1. Удалите throws Exception из метода getKey().
2. Измените метод getKey(), добавив try-catch блок для обработки исключений.
3. Добавьте цикл к getKey() таким образом, чтобы пользователь получил еще один шанс на ввод значение ключа

**Замечания:**

Инструкция throw очень аналогична инструкции return – она прекращает выполнение метода, дальше мы не идем. Если мы нигде не ставим catch, то у нас бросок Exception очень похож на System.exit() – система вырубается. Но мы в любом месте можем поставить catch и, таким образом, предотвратить поломку кода.

**Выводы:**

Фактически при работе с исключениями весь материал делится на две части: *синтаксис* (ответ на вопрос, что компилятор пропустит, а что нет) и *семантика* (вопрос, как лучше делать) исключений. В отличие от вариантов с for, while, switch, использование исключений  – более сложный механизм. Но он сложен не синтаксически, а семантически, по своему подходу. То есть при генерации исключений нужно думать о том - не как правильно его использовать, а *с каким смыслом* его использовать. То есть вопрос, в каких ситуациях стоит ли ломать систему и где, а в каких ситуациях ее восстанавливать.

В хорошей инженерной системе каждый любой модуль всегда проверяет все аргументы на входе.

Можно еще сказать что существует иерархия различных способов *прекращения выполнения некоего действия (или ряда действий) в виде участка кода*  и эта иерархия классифицирует возможные действия по мощности используемого оператора: continue, break, return, throw.

* **continue**прекращает выполнение *данной итерации* в цикле;
* **break**прекращает выполнение *данного блока (например цикла)*;
* **return**– это инструкция выхода*из данного метода (например прекращение выполнения функции)*;
* **throw**– еще более сильная инструкция, она прекращает выполнение данного метода *и метода, который его вызвал*. Так-как исключения вообще-то позволяют сломать весь стек работы программы.