Лекция 6

Знакомство с исключениями: try, catch, throws, multy-catch.

Stack

- Представьте себе стопку бумаг — деловых поручений для некоторого исполнителя. Сверху на стопку можно класть новое задание, и с верха стопки задание можно брать. При таком подходе задания будут исполняться не по порядку поступления. Задание, положенное самым последним, будет взято исполнителем самым первым. Такая структура элементов коллекции называется **стеком** — стопкой.

- В Java для этого есть специальная коллекция – Stack. Это коллекция, у которой есть методы «добавить элемент» и «взять(достать/забрать) элемент». Первым будет взят элемент, добавленный самым последним.

StackTrace

- Представьте себе, что в Java функция А вызвала функцию Б, а та вызвала функцию В, а та, в свою очередь, функцию Г. Так вот, чтобы выйти из функции Б, нужно сначала выйти из функции В, а для этого выйти из функции Г. Это очень похоже на стек.

- В стопке тоже, чтобы добраться до какого-то листка с заданием, надо довыполнить все задания, которые положили сверху.

- Стек — это набор элементов. Как листы в стопке. Чтобы взять третий сверху лист, надо сначала взять второй, а для этого взять первый. Класть и брать листы можно всегда, но всегда взять можно только самый верхний.

- С вызовом функций то же самое. Функция А вызывает функцию Б, а та вызывает функцию В. И чтобы выйти из А, надо сначала выйти из Б, а для этого надо выйти из В.

- Стек сведется к «взять можно только самый последний положенный лист», «выйти можно только из последней функции, в которую зашли».

- Так вот — последовательность вызовов функций — это и есть «стек вызовов функций», он же просто «стек вызовов». Функция, вызванная последней, должна завершиться самой первой.

Получение и вывод текущего стека вызовов:

```
public class ExceptionExample
{
  public static void main(String[] args)
  {
    method1();
  }

  public static void method1()
  {
    method2();
  }

  public static void method2()
  {
    method3();
  }

  public static void method3()
  {
    StackTraceElement[] stackTraceElements = Thread.currentThread().getStackTrace();
    for (StackTraceElement element : stackTraceElements)
    {
      System.out.println(element.getMethodName());
    }
  }
}
```

Результат:

```
getStackTrace
method3
method2
method1
main
```

- Java-машина ведет запись всех вызовов функций. У нее есть для этого специальная коллекция – стек (Stack). Когда одна функция вызывает другую, Java-машина помещает в этот стек новый элемент StackTraceElement. Когда функция завершается этот элемент удаляется из стека. Таким образом, в этом стеке всегда хранится актуальная информация о текущем состоянии «стека вызовов функций».

- Каждый StackTraceElement содержит информацию о методе, который был вызван. В частности можно получить имя этого метода с помощью функции getMethodName.

- В предыдущем примере:
  1) Получаем «стек вызовов»:
  2) Проходимся по нему с помощью цикла **for-each**.
  3) Печатаем в **System.out** имена методов.

Знакомство с исключениями

- Исключения – это специальный механизм для контроля над ошибками в программе. Вот примеры ошибок, которые могут возникнуть в программе:
  1. Программа пытается записать файл на заполненный диск.
  2. Программа пытается вызвать метод у переменной, которая хранит ссылку – null.
  3. Программа пытается разделить число на 0.

- Все эти действия приводят к возникновению ошибки. Обычно это приводит к закрытию программы — продолжать выполнять дальше код не имеет смысла.

- Потом разработчики придумали интересный ход: каждая функция возвращала статус своей работы. 0 означал, что она отработала как надо, любое другое значение – что произошла ошибка: это самое значение и было кодом ошибки.

- Но был у такого подхода и минус. После каждого(!) вызова функции нужно было проверять код (число), который она вернула. Во-первых, это было неудобно: код по обработке ошибок исполнялся редко, но писать его нужно было всегда. Во-вторых, функции часто сами возвращают различные значения – что делать с ними?

Механизм обработки ошибок

- 1. Когда возникает ошибка, Java-машина создаёт специальный объект – exception – исключение, в который записывается вся информация об ошибке. Для разных ошибок есть разные исключения.

- 2 Затем это «исключение» приводит к тому, что программа тут же выходит из текущей функции, затем выходит из следующей функции, и так пока не выйдет из метода main. Затем программа завершается. Еще говорят, что Java-машина «раскручивает назад стек вызовов».

- Есть способ перехватить исключение. В нужном месте, для нужных нам исключений можно написать специальный код, который будет перехватывать эти исключения и что-то делать.

- Для этого есть специальная конструкция **try-catch**.

- Пример программы, которая перехватывает исключение – деление на 0. И продолжает работать.

```
public class ExceptionExample2
{
   public static void main(String[] args)
   {
      System.out.println("Program starts");

      try
      {
         System.out.println("Before method1 calling");
         method1();
         System.out.println("After method1 calling. Never will be shown");
      }
      catch (Exception e)
      {
        System.out.println("Exception has been caught");
      }

      System.out.println("Program is still running");
   }

   public static void method1()
   {
      int a = 100;
      int b = 0;
      System.out.println(a / b);
   }
}
```

- Вывод на экран:

«Program starts»
«Before method1 calling»
«Exception has been caught»
«Program is still running»

- В строчке System.out.println(a / b); было деление на ноль. Это привело к возникновению ошибки – исключения. Java-машина создала объект ArithmeticException с информацией об ошибке. Этот объект является исключением.

- Внутри метода method1 возникло исключение. И это привело к немедленному завершению этого метода. Оно привело бы и к завершению метода **main**, если бы не было блока **try-catch.**

- Если внутри блока **try** возникает исключение то, оно захватывается в блоке **catch**. Остаток кода в блоке try, не будет исполнен, а сразу начнётся исполнение блока **catch**.

- Этот код работает так:
  1. Если внутри блока **try** возникло исключение, то код перестаёт исполняться, и начинает исполняться блок **catch**.
  2. Если исключение не возникло, то блок try исполняется до конца, а **catch** никогда так и не начнёт исполняться.

- Представьте, что после вызова каждого метода мы проверяем: завершился ли только что вызванный метод сам по себе или в результате исключения. Если исключение было, тогда мы переходим на исполнение блока catch, если он есть, и захватываем исключение. Если блока catch нет, то завершаем и текущий метод. Тогда такая же проверка начинается в том методе, который вызвал нас.

- Все исключения – это классы, унаследованные от класса Exception. Мы можем перехватить любое из них, указав в блоке **catch** его класс, или все сразу, указав общий родительский класс — Exception. Затем из переменной e (эта переменная хранит ссылку на объект исключения), можно получить всю необходимую информацию о возникшей ошибке.

- Если в вашем методе возникнут разные исключения, можно обрабатывать их по-разному.

- Блок try может содержать несколько блоков catch, каждый из которых будет захватывать исключения своего типа.

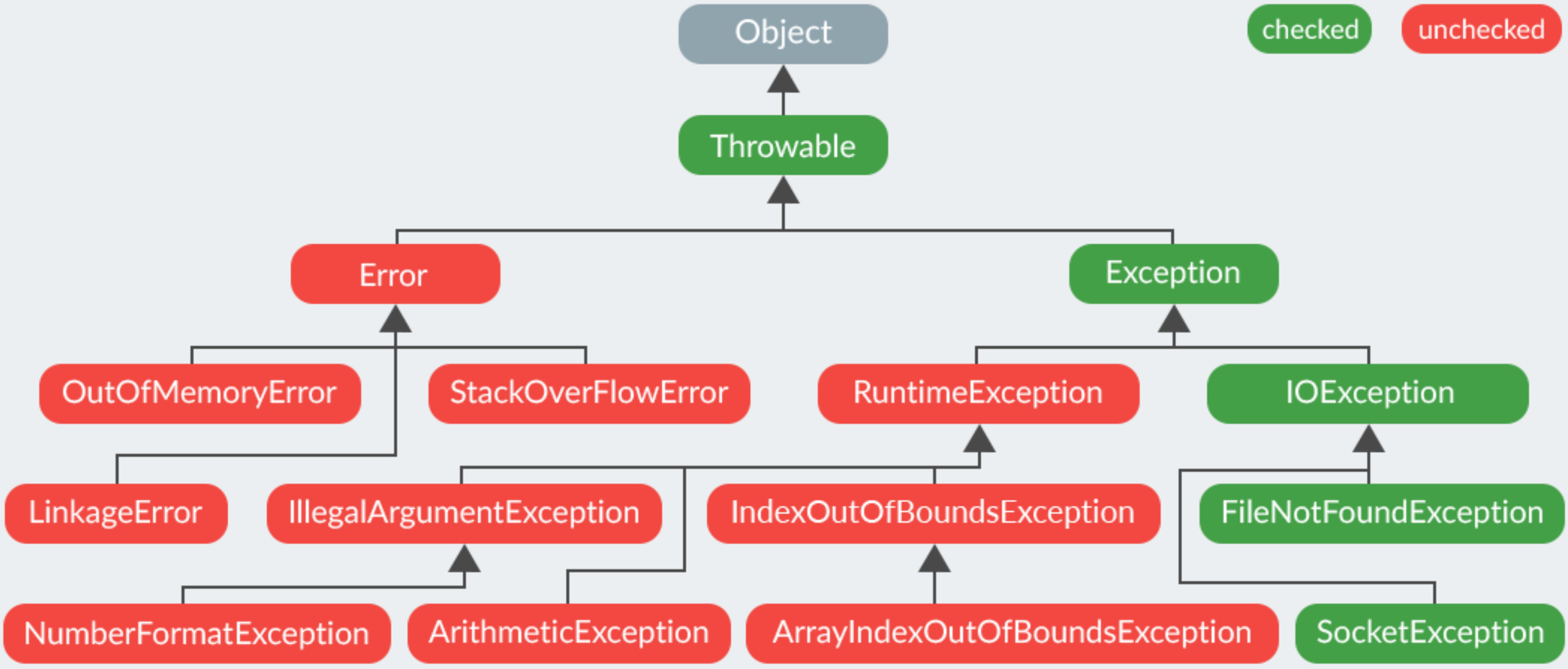
```
public class ExceptionExample3
{
   public static void main(String[] args)
   {
      System.out.println("Program starts");

      try
      {
         System.out.println("Before method1 calling");
         method1();
         System.out.println("After method1 calling. Never will be shown ");
      }
      catch (NullPointerException e)
      {
        System.out.println("Reference is null. Exception has been caught");
      }
      catch (ArithmeticException e)
      {
         System.out.println("Division by zero. Exception has been caught");
      }
      catch (Exception e)
      {
         System.out.println("Any other errors. Exception has been caught");
      }

      System.out.println("Program is still running");
   }

   public static void method1()
   {
      int a = 100;
      int b = 0;
      System.out.println(a / b);
   }
}
```

Типы исключений



- В Java все исключения делятся на два типа – контролируемые/проверяемые (checked) и неконтролируемые/непроверяемые (unchecked): те, которые перехватывать обязательно, и те, которые перехватывать не обязательно. По умолчанию – все исключения обязательно нужно перехватывать.

- Если в методе выбрасываются (возникают) исключения ClassNotFoundException и FileNotFoundException, программист обязан указать их в сигнатуре метода (в заголовке метода). Это checked исключения.

Примеры

```
public static void method1() throws ClassNotFoundException, FileNotFoundException

public static void main() throws IOException

public static void main() //не выбрасывает никаких исключений
```

Примеры проверяемых (checked) исключений

```
public static void main(String[] args)
{
   method1();
}

public static void method1() throws FileNotFoundException, ClassNotFoundException
{
   //тут кинется исключение  FileNotFoundException, такого файла нет
   FileInputStream fis = new FileInputStream("C2:\badFileName.txt");
}
```

- Этот пример не скомпилируется, т.к. метод **main** вызывает метод method1(), который выкидывает исключения, обязательные к перехвату.

- Чтобы программа скомпилировалась, метод, который вызывает method1, должен сделать две вещи: или перехватить эти исключения или пробросить их дальше (тому, кто его вызвал), указав их в своём заголовке.

- Если в методе main вызвать метод какого-то объекта, в заголовке которого прописано throws FileNotFoundException, ... то надо сделать одно из двух:
  1) Перехватывать исключения FileNotFoundException, ...
  Придется обернуть код вызова опасного метода в блок **try-catch**
  2) Не перехватывать исключения FileNotFoundException, ...
  Придется добавить эти исключения в список **throws** своего метода **main**.

Способ 1: просто пробрасываем исключение выше (вызывающему):

```
public static void main(String[] args)  throws FileNotFoundException, ClassNotFoundException
{
   method1();
}

public static void method1() throws FileNotFoundException, ClassNotFoundException
{
   //тут кинется исключение  FileNotFoundException, такого файла нет
   FileInputStream fis = new FileInputStream("C2:\badFileName.txt");
}
```

Способ 2: перехватываем исключение:

```
public static void main(String[] args)
{
   try
   {
      method1();
   }
   catch(Exception e)
   {
   }
}

public static void method1() throws FileNotFoundException, ClassNotFoundException
{
   //тут кинется исключение  FileNotFoundException, такого файла нет
   FileInputStream fis = new FileInputStream("C2:\badFileName.txt");
}
```

Не обрабатываем исключения – нужно пробросить их дальше, тому, кто знает как

```
public static void method2() throws FileNotFoundException, ClassNotFoundException
{
   method1();
}
```

Обрабатываем одно исключение, второе – пробрасываем:

```
public static void method3() throws ClassNotFoundException
{
  try
   {
      method1();
   }
   catch (FileNotFoundException e)
   {
      System.out.println("FileNotFoundException has been caught.");
   }
}
```

Перехватываем оба – ничего не пробрасываем:

```
public static void method4()
{
   try
   {
      method1();
   }
   catch (FileNotFoundException e)
   {
      System.out.println("FileNotFoundException has been caught.");
   }
   catch (ClassNotFoundException e)
   {
      System.out.println("ClassNotFoundException has been caught.");
   }
}
```

- Но есть вид исключений – это RuntimeException и классы, унаследованные от него. Их перехватывать не обязательно. Это unchecked исключения. Считается, что это трудно прогнозируемые исключения и предсказать их появление практически невозможно. С ними можно делать все то же самое, но указывать в **throws** их не нужно.

Код с использованием исключений

```
class ExceptionExampleOriginal
{


   public static void main(String[] args)
   {
      System.out.println("main begin");
      try
      {
         System.out.println("main before call");
        method1();
         System.out.println("main after call");
      }
      catch (RuntimeException e)
      {


         String s = e.getMessage();
         System.out.println(s);
      }
      System.out.println("main end");
   }


   public static void method1()
   {
      System.out.println("method1 begin");
      method2();

      System.out.println("method1 end");
   }

   public static void method2()
   {
    System.out.println("method2");
    String s = "Message: Unknown Exception";
    throw new RuntimeException(s);

   }
}
```

Примерная расшифровка

```
public class ExceptionExample
{
   private static Exception exception = null;

  public static void main(String[] args)
   {
      System.out.println("main begin");

      System.out.println("main before call");
     method1();
     if (exception == null)
     {
        System.out.println("main after call");
     }
     else if (exception instanceof RuntimeException)
     {
        RuntimeException e = (RuntimeException) exception;
        exception = null;
        String s = e.getMessage();
        System.out.println(s);
     }
     System.out.println("main end");
   }

   public static void method1()
   {
     System.out.println("method1 begin");
     method2();
     if (exception != null) return;
     System.out.println("method1 end");
   }

   public static void method2()
   {
     System.out.println("method2");
     String s = "Message: Unknown Exception";
     exception = new RuntimeException(s);
     return;
   }
}
```

- В примере слева по цепочке вызываем несколько методов. В method2 специально создаем и выкидываем исключение (инициируем ошибку).

- В примере справа показано, что примерно при этом происходит.

- Посмотрите на method2. Создание исключения превратилось вот во что: создали объект типа RuntimeException, сохранили его в специальную переменную exception и тут же вышли из метода – return.

- В методе method1, после вызова method2 стоит проверка – есть исключение или нет, если исключение есть, тогда метод method1 тут же завершается. Такая проверка неявно производится после вызова каждого(!) метода в Java.

- В колонке справа в методе main написано, что примерно происходит при перехвате исключения с помощью конструкции try-catch. Если исключения не было, то все продолжает работать, как и запланировано. Если исключение было, и оно было такого типа, как указано в catch, тогда мы его обрабатываем.

- Посмотрите на последнюю строку throw new RuntimeException(s). Таким способом мы создаем и кидаем исключение. Так делать не нужно. Это только для примера.

- А с помощью команды «a instanceof B» проверяем, имеет ли объект a тип B. Т.е. имеет ли объект, который хранится в переменной exception, тип RuntimeException. Это логическое условие.

Встроенные исключения Java

- Существуют несколько готовых системных исключений. Большинство из них являются подклассами типа **RuntimeException** и их не нужно включать в список **throws**. Вот небольшой список непроверяемых исключений.
  - ArithmeticException - арифметическая ошибка, например, деление на нуль
  - ArrayIndexOutOfBoundsException - выход индекса за границу массива
  - ArrayStoreException - присваивание элементу массива объекта несовместимого типа
  - ClassCastException - неверное приведение
  - EnumConstantNotPresentException - попытка использования неопределённого значения перечисления
  - IllegalArgumentException - неверный аргумент при вызове метода
  - IllegalMonitorStateException - неверная операция мониторинга
  - IllegalStateException - некорректное состояние приложения
  - IllegalThreadStateException - запрашиваемая операция несовместима с текущим потоком
  - IndexOutofBoundsException - тип индекса вышел за допустимые пределы
  - NegativeArraySizeException - создан массив отрицательного размера
  - NullPointerException - неверное использование пустой ссылки
  - NumberFormatException - неверное преобразование строки в числовой формат
  - SecurityException - попытка нарушения безопасности
  - StringIndexOutOfBounds - попытка использования индекса за пределами строки
  - TypeNotPresentException - тип не найден
  - UnsupportedOperationException - обнаружена неподдерживаемая операция

- Список проверяемых системных исключений, которые можно включать в список **throws**. (Можно, но на практике не нужно)
  - ClassNotFoundException - класс не найден
  - CloneNotSupportedException - попытка клонировать объект, который не реализует интерфейс **Cloneable**
  - IllegalAccessException - запрещен доступ к классу
  - InstantiationException - попытка создать объект абстрактного класса или интерфейса
  - InterruptedException - поток прерван другим потоком
  - NoSuchFieldException - запрашиваемое поле не существует
  - NoSuchMethodException - запрашиваемый метод не существует
  - ReflectiveOperationException - исключение, связанное с рефлексией

Как работает множественный catch

- При возникновении исключения в блоке **try**, выполнение программы передаётся на первый **catch**.

- Если тип, указанный внутри круглых скобок блока catch, совпадает с типом объекта-исключения, то начинается выполнение кода внутри блока {}. Иначе переходим к следующему **catch**. Там проверка повторяется.

- Если блоки **catch** закончились, а исключение так и не было перехвачено, то оно выбрасывается дальше, а текущий метод аварийно завершается.

- Но в реальности все немного сложнее. Дело в том, что классы можно наследовать друг от друга. И если класс «Корова» унаследовать от класса «Животное», то объект типа «Корова» можно хранить не только в переменной типа «Корова», но и в переменной типа «Животное».

- Т.к. все исключения унаследованы от классов **Exception** или **RuntimeException** (который тоже унаследован от **Exception**), то их все можно перехватить командами catch **(Exception** e) или **catch (RuntimeException e)**.

- Отсюда два вывода. Во-первых, с помощью команды catch(Exception e) можно перехватить любое исключение вообще. Во-вторых — порядок блоков catch имеет значение.

- Возникший при делении на 0 ArithmeticException будет перехвачен во втором catch.

```
try
{
   System.out.println("Before method1 calling.");
   int a = 1 / 0;
   System.out.println("After method1 calling. Never will be shown.");
}
catch (NullPointerException e)
{
   System.out.println("Reference is null. Exception has been caught.");
}
catch (ArithmeticException e)
{
   System.out.println("Division by zero. Exception has been caught.");
}
catch (Exception e)
{
   System.out.println("Any other errors. Exception has been caught.");
}
```

- В примере ниже возникший ArithmeticException будет перехвачен в первом catch, т.к. классы всех исключений унаследованы от Exception. Т.е. Exception захватывает любое исключение.

```
try
{
   System.out.println("Before method1 calling.");
   int a = 1/0;
   System.out.println("After method1 calling. Never will be shown.");
}
catch (Exception e)
{
   System.out.println("Any other errors. Exception has been caught.");
}
catch (NullPointerException e)
{
   System.out.println("Reference is null. Exception has been caught.");
}
catch (ArithmeticException e)
{
   System.out.println("Divided by zero. Exception has been caught.");
}
```

- В примере ниже исключение ArithmeticException не будет перехвачено, а будет выброшено дальше в вызывающий метод.

```
try
{
   System.out.println("Before method1 calling.");
   int a = 1/0;
   System.out.println("After method1 calling. Never will be shown.");
}
catch (NullPointerException e)
{
   System.out.println("Reference is null. Exception has been caught.");
}
```

Задачи

1. Написать пять методов, которые вызывают друг друга. Каждый метод должен возвращать свой StackTrace.

```
public class Solution {
	public static void main(String[] args) {
		method1();
	}

	public static StackTraceElement[] method1() {
		method2();
		//напишите тут ваш код
	}

	public static StackTraceElement[] method2() {
		method3();
		//напишите тут ваш код
	}

	public static StackTraceElement[] method3() {
		method4();
		//напишите тут ваш код
	}

	public static StackTraceElement[] method4() {
		method5();
		//напишите тут ваш код
	}

	public static StackTraceElement[] method5() {
		//напишите тут ваш код
	}
}
```

2. Написать пять методов, которые вызывают друг друга.
Каждый метод должен возвращать **имя метода**, вызвавшего его (текущий метод), полученное с помощью StackTrace. method1 вызван в main - надо вывести main.

3. Написать пять методов, которые вызывают друг друга. Метод должен вернуть **номер строки кода**, из которого вызвали этот метод. Воспользуйтесь функцией: element.getLineNumber().

Подсказка: напишите программу, посмотрите, какое исключение возникает, а потом поменяйте код и перехватите его.

Программа должна отлавливать исключения конкретного типа, а не все возможные (Exception).

4. **Перехватить** исключение, возникающее при выполнении кода: int a = 42 / 0; Вывести на экран тип перехваченного исключения.

```
public class Solution {
    public static void main(String[] args) {
        //напишите тут ваш код

        int a = 42 / 0;

        //напишите тут ваш код
    }
}
```

5. **Перехватить** исключение (и вывести его на экран), указав его тип, возникающее при выполнении кода:
String s = null;
String m = s.toLowerCase();