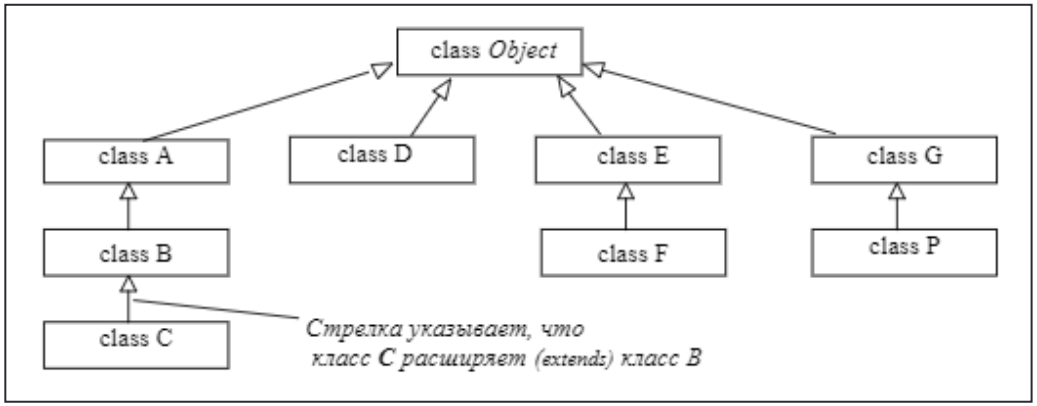
Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования «Чувашский государственный университет имени И.Н. Ульянова»

НАСЛЕДОВАНИЕ. ОШИБКИ И ИСКЛЮЧЕНИЯ

Назарова Ольга Васильевна

Все классы Java являются наследниками каких -то других классов, как показано в иерархии классов на рис. 1.



Наследование одного класса от другого можно записать так:

```
class C extends B{… // класс B – родитель, класс C – потомок
    содержимое класса
}
```

Класс-потомок (дочерний класс) наследует все методы класса-предка (родительского класса), а также расширяет (**extends**) родительский класс, 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**.

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.

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

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

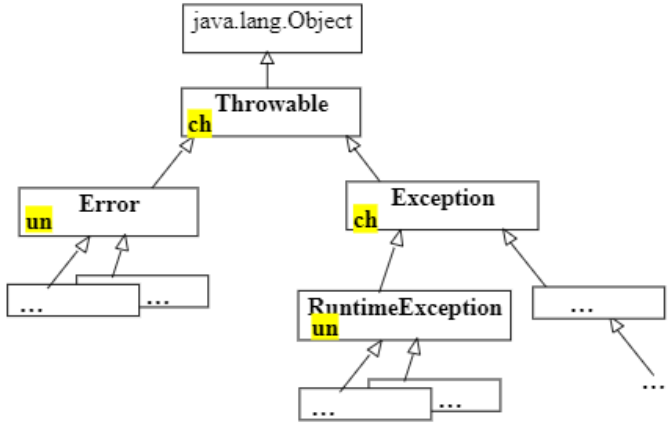
```
13  public class Except {
14
15      /**...*/
18
19      static  int m1(){
20        return m2()*10;
21      }
22      static int m2(){
23          Random rn= new Random(); // создание объекта класса Rando
24          int     x=rn.nextInt(2);
25          return 10/x;
26      }
27      public static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

```
Вывод – meyhodsExcep2 (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**

Рис. 2

Чтобы разбираться в исключительных ситуациях и уметь корректно реагировать на них в программе, следует ознакомиться с иерархией наследования классов исключений. На рис. 3 представлены четыре базовых класса исключений: Throwable, Error, Exception, RuntimeException – от них наследуются все остальные классы исключений Java.



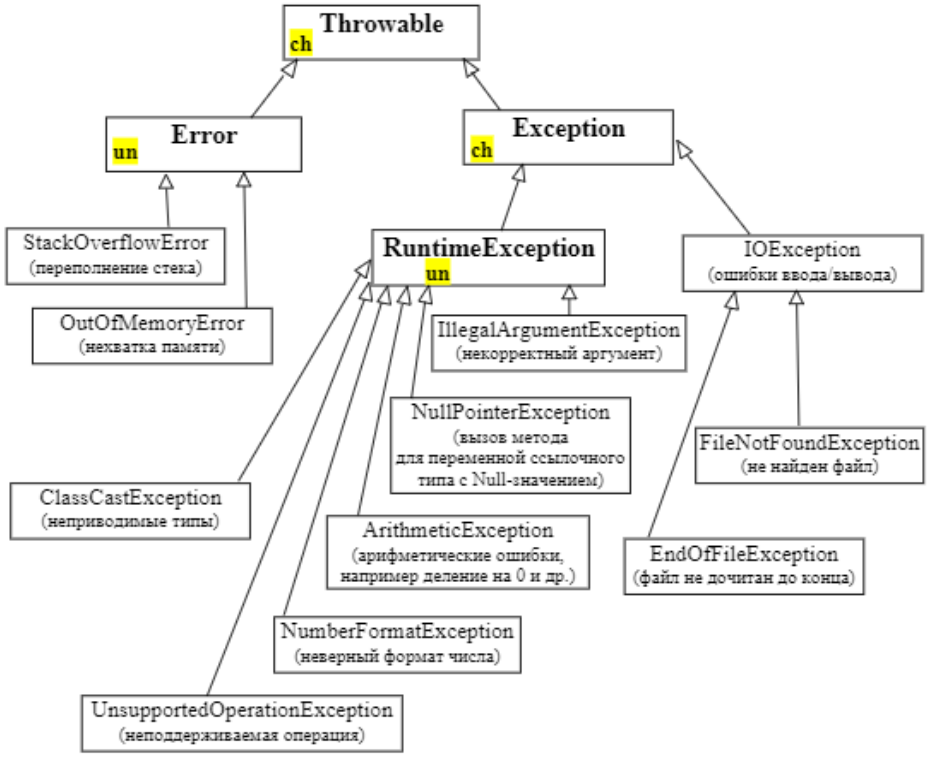
где **ch** – checked; **un** – unchecked

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. 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.



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.

ПРИМЕР: Сгенерировано и перехвачено RuntimeException.

```
public static void main(String[] args)  {
    try {
       System.out.println("0");
       throw   new RuntimeException("Непроверяемая ошибка");
          «брошено» исключение        создан экземпляр RuntimeException с сообщением

    } catch (RuntimeException e) {   // исключение перехвачено
        System.out.println("1  "+ e);  // исключение обработано
     }
    System.out.println("2");
 }
```

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 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");
     }
  } }
```