Класс Object

-это суперкласс всех классов JAVA, поэтому ссылочная переменная класса **Object** может ссылаться на объект любого класса (в том числе на любой массив).

Класс Object определяет методы, которые доступны в любом объекте.

Класс Object определяет методы:

1) **Object clone( )** – создает новый объект – копию.
2) **boolean equals(Object object)** – проверяет объекты на равенство.
3) **void finalize ( )** – завершающие действия, вызывается до сборщика мусора.
4) **Class getClass( )** – получает класс объекта во время выполнения.

Класс Object

5) **int hashCode( )** – возвращается хэш-код, связанный с вызовом объекта.
6) **void notify( )** – возобновляет выполнение потока, ожидающего на объекте вызова.
7) **void notifyAll ( )** – возобновляет выполнение всех потоков, ожидающих на объекте вызовов
8) **String toString( )** – возвращает строку, которая описывает объект.
9) **void wait()**
void wait (long milliseconds)
void wait (long milliseconds, int nanoseconds) – ждет выполнения на другом потоке.

Метод **toString()** возвращает строку, содержащую описание объекта и вызывается автоматически, когда объект выводится с помощью метода println().

Обработка исключений

Исключение - это **ошибка**, возникающая во время выполнения последовательности кода (ошибка времени выполнения).

Исключение в JAVA – это **объект**, который описывает исключительную (ошибочную) ситуацию, возникающую в части программного кода.

Когда исключительная ситуация возникает, создается **объект**, представляющий (описывающий) исключение, который **выбрасывается** в методе, вызвавшем ошибку.

Метод может либо обработать исключение самостоятельно, либо пропустить его. В обоих случаях, в некоторой точке исключение перехватывается и обрабатывается.

Основы обработки исключений

Исключения могут генерироваться системой времени выполнения JAVA, либо они могут быть сгенерированы «вручную» в коде.

Исключения, которые возбуждает JAVA, имеют отношение к фундаментальным ошибкам, которые нарушают правила языка JAVA, либо ограничения системы выполнения JAVA.

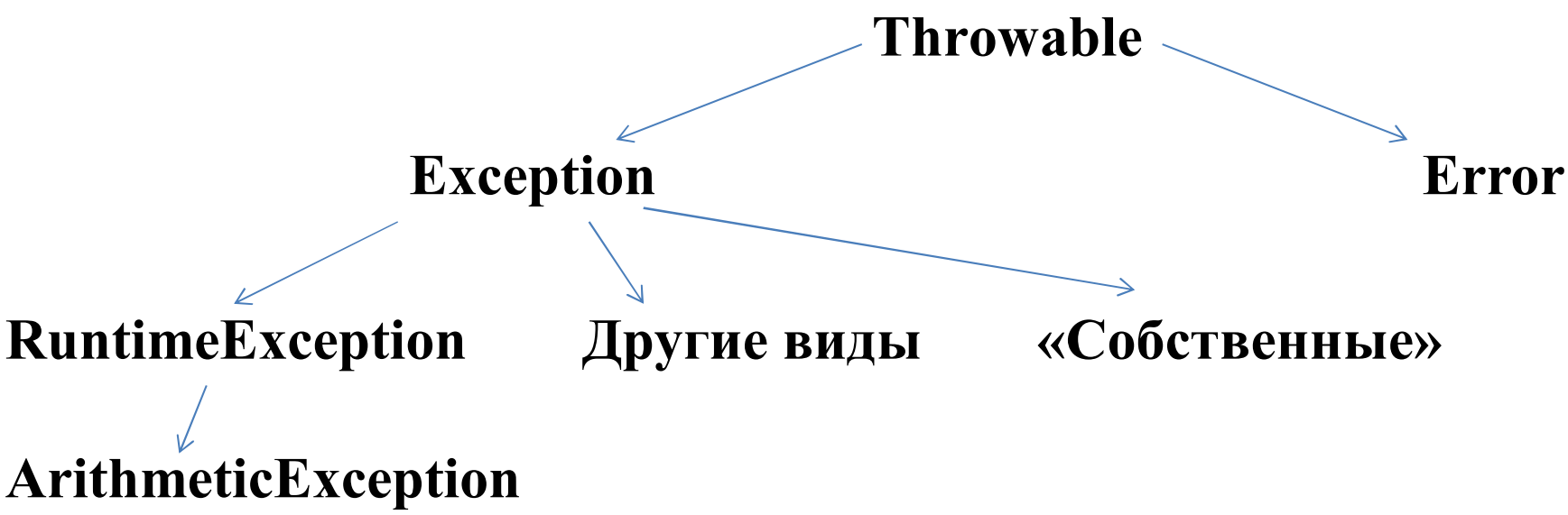
Вручную сгенерированные исключения обычно применяются для того, чтобы сообщить о некоторых ошибочных ситуациях тому, кто вызвал данный метод.

Сгенерированные системой исключения автоматически выбрасываются системой времени выполнения JAVA.

Типы исключений

Все типы исключений являются подклассами класса **Throwable**. Класс Throwable - вершина иерархии классов исключений.

Два подкласса Throwable разделяют все исключения на две отдельные ветви. **java.lang.Throwable**



Типы исключений

1) Класс **Exception** (используется для исключительных ситуаций, которые пользовательская программа должна перехватывать, а также для создания «собственных» исключений).

Класс **Exception** имеет подкласс **RuntimeException**. Исключения **RuntimeException** автоматически определяются для программ (например, деление на ноль и ошибочная индексация массивов).

2) Класс **Error** определяет исключения, вызов которых не ожидается при нормальном выполнении программы. Исключения типа **Error** используются системой времени выполнения JAVA для обозначения ошибок, происходящих внутри самой системы времени выполнения (JVM), например, переполнение стека. Ошибки не относятся к пользовательской программе.

Подклассы RunTimeExeption, определенные в java.lang
(не требуют обозначения через throws в заголовке метода)

ArithmeticException - арифметическая ошибка (н-р,деление на 0).

ArrayIndexOutOfBoundsException - выход индекса за границу массива.

IllegalArgumentException - неверный аргумент при вызове метода.

IndexOutOfBoundsException - некоторый тип индекса вышел за допустимые пределы.

NegativeArraySizeException - отрицательный размер массива.

NullPointerException – обращение к несуществующему объекту.

NumberFormatException - неверное преобразование строки в числовой формат.

StringIndexOutOfBounds - Попытка использования индекса за пределами строки.

Необработанные исключения

```
class Exc {
    public static void main(String args[]) {
        int d =0;
        int a = 42 / d;          //прерывание работы Exc
                        } }      //и выброс исключения
```

Попытка деления на ноль - JVM конструирует объект исключения и выбрасывает его. Это прерывает выполнение Exc, т.к. исключение возбуждено и должно быть перехвачено обработчиком исключений.

Если нет блока **try{ }catch{ }** в программе, то исключение перехватывается обработчиком по умолчанию. Обработчик по умолчанию отображает строку, описывающую исключение, печатает **трассировку стека** от точки возникновения исключения и прерывает программу.

Трассировка стека

java.lang.ArithmeticException: / by zero
        at Exc.main(Exc.java:4)

Файл **Exc.java**, строка **4**, ошибка **/by zero**, тип исключения **ArithmeticException**.

```
class Exc1 {

   static  void  trassa( ) {
           int d = 0;
           int a = 10 / d;  }

   public  static  void  main(String args[]) {
            Exc1.trassa();   } }
```

java.lang.ArithmeticException: / by zero
        at Exc1.trassa(Exc1.java:4)
        at Exc1.main(Exc1.java:6)

Основы обработки исключений

Обработка исключений JAVA осуществляется пятью ключевыми словами: **try, catch, throw, throws и finally.**
Общая форма блока обработки исключений:

```
try{
    // блок кода, в котором отслеживаются ошибки
                    }
catch (ExceptionType1   ExcOb) {
     // обработчик исключений типа ExceptionTypel
…                                              }
catch (ExceptionTypeN ExcOb) {
 // обработчик исключений типа ExceptionTypeN
                                              }
finally {
   // блок кода, который должен быть обязательно
   //выполнен после завершения блока try
         }
```

Использование try и catch

Используется, если необходимо обрабатывать исключения самостоятельно «вручную».

1. Предоставляет возможность исправить ошибку.
2. Предотвращает автоматическое прерывание выполнения программы.

Чтобы обрабатывать ошибки, возникающие во время выполнения программы, нужно поместить код, в котором может возникнуть ошибка, внутрь блока **try{ }**. За блоком **try{ }**следует включить конструкцию **catch{ }**, которая определяет, перехватывает и обрабатывает исключение по типу.

Использование try и catch

```
class Exc2 {
  public static void main(String args[]) {
    int d, a;
    try {     // мониторинг кода
        d = 0;
        a = 42 / d;
        System.out.println("Это не будет напечатано");  }
   catch(ArithmeticException   e) {      // перехват ошибки
      System.out.println("Деление на ноль");   }
   System.out.println("После оператора catch");
} }
```

Деление на ноль
После оператора catch

Использование try и catch

После того, как блок **catch{ }** будет выполнен, управление передается на строку программы, следующую за всем механизмом **try{ } catch{ }**.

Область действия **catch** не распространяется на те операторы, которые идут перед оператором **try**.

Оператор **catch** не может перехватить исключение, возбужденное другим оператором **try** (кроме случаев вложенных конструкций **try**).

Операторы, которые защищены блоком **try**, должны быть заключены в фигурные скобки.

Целью **catch** является продолжение работы программы, будто ошибка не произошла.

Использование try и catch

```
import  java.util.Random;

 class ErrorExc{

  public static void main(String args[]) {
        int a=0, b=0, c=0;
        Random r = new Random();

      for(int i=0; i<32000; i++) {
            try {
                b=r.nextInt( );
                c=r.nextInt( );
                a =12345 / (b/c);  }

            catch (ArithmeticException e) {
                  System.out.println("Деление на ноль");
                  a = 0;                              }
      System.out.println("a: " + a);
   }}
```

Отображение описания исключения

Класс **Throwable** переопределяет метод **public String toString().** Метод **toString()** возвращает строку, содержащую описание исключения. Отобразить это описание можно с помощью метода println(), передав исключение в виде аргумента. Например, блок catch из предыдущего примера может быть переписан следующим образом:

```
catch (ArithmeticException e) {
    System. out. println ("Исключение: " + e) ;
     a = 0;   }
```

Исключение: java.lang.ArithmeticException: / by zero

Множественные операторы catch

В некоторых случаях один фрагмент кода может инициировать более одного исключения.

Можно определить два или более операторов **catch**, каждый для перехвата своего типа исключений.

Когда выбрасывается исключение, каждый оператор **catch** проверяется по порядку, и первый из них, чей тип соответствует исключению, выполняется.

После того, как выполнится один из операторов **catch**, все остальные пропускаются, и выполнение программы продолжается с места, следующего за блоком **try{}catch{}**.

Множественные операторы catch

```
class MultiCatch {
  public static void main(String args[]) {
      try {
          int a = args.length;
          System.out.println("a = " + a);
          int b = 42 / a;
          int c[ ] = { 1 };
            c[42] = 99;          }

    catch(ArithmeticException   e) {
          System.out.println("Деление на 0: " + e);   }

   catch(ArrayIndexOutOfBoundsException  e) {
          System.out.println("Ошибка индекса массива: " + e);  }

   System.out.println("После блока try/catch.");
 } }
```

java MultiCatch abc или **java MultiCatch**

Множественные операторы catch

При использовании множественных операторов catch, важно помнить, что подклассы исключений должны следовать перед любыми их суперклассами. Иначе оператор catch, который использует тип исключения суперкласса, будет перехватывать все исключения этого суперкласса и всех его подклассов. В данном случае исключение типа подкласса никогда не будет обработано, и возникает ошибка компиляции, т.к. в JAVA недостижимый код является ошибкой.

Множественные операторы catch

```
class SuperCatch {
   public static void main(String args[ ]) {
      try{
           int a = 0;
           int b = 42 / a;  }
       catch(Exception e)  {
           System.out.println ("Общий перехват Exception ");  }
      catch(ArithmeticException e) {
           System.out.println("Это никогда не выполнится");  }
   }}
```

Вложенные операторы try

Один оператор try может находиться внутри блока другого try. Если вложенный оператор try не имеет обработчика catch для определенного исключения, то проверяются на соответствие обработчики catch следующего (внешнего) блока try. Это продолжается до тех пор, пока не будет найден подходящий оператор catch, либо пока не будут проверены все уровни вложенных try. Если подходящий оператор catch не будет найден, то исключение обработает система времени выполнения JAVA.

Вложенные операторы try

```
class NestTry{
  public static void main(String args[ ]){
     try {
        int a = args.length;      //java  NestTry  abc или java  NestTry
        int b = 42 / a;
        System.out.println("a = " + a);

           try{
              if(a == 1)  a = a  / (a - a);  // деление на ноль
              if(a == 2) {
                    int c [ ] = { 1 };
                       c[42] = 99;         }

           catch(ArrayIndexOutOfBoundsException  e) {
            System.out.println("Индекс за пределами массива: " + e); }

   catch(ArithmeticException   e) {
            System.out.println(“Деление на 0: " + e); }
 }  }
```

throw

Оператор **throw** используется для выброса исключения.
Общая форма**:** **throw Объект_Throwable**;
Объект_Throwable должен быть объектом типа Throwable, либо подклассом Throwable (Exception и подклассы Exception).

Два способа получить объект Throwable:
1) параметр в операторе catch ;
2) создание объекта с помощью new.

Выполнение программы останавливается после оператора throw, любые последующие операторы не выполняются. Далее происходит поиск блока try с оператором catch соответствующего типа исключения. Если соответствие найдено, управление передается этому оператору. Если же нет, проверяется следующий внешний блок try. Если нет подходящего по типу оператора catch, то обработчик исключений по умолчанию прерывает программу и печатает трассировку стека.

throw

```
class ThrowDemo {
  static void proc() {
      try {
          throw new NullPointerException ("demo") ;  }
      catch(NullPointerException  e) {
          System.out.println("Перехвачено внутри  proc()");
          throw  e;     }       // повторный выброс
    }
 public static void main(String args[]) {
         try {
               proc();    }
        catch(NullPointerException   e) {
             System.out.println(“Повторный перехват: “+ e);    }
  } }
```

throws

Если метод может выбросить исключение, которое он не обрабатывает, то этот метод должен сообщить вызывающим его методам об этом исключении.

В название метода добавляется конструкция **throws**.

throws перечисляет типы исключений, которые метод может выбрасывать. Это необходимо использовать для всех исключений, кроме **Error, RuntimeException**, либо их подклассов.

Все остальные исключения, которые может возбуждать метод, должны быть объявлены в конструкции **throws**.

Общая форма объявления метода с использованием **throws**:

тип ИмяМетода (список параметров**) throws СписокИсключений**
{
// тело метода **}**

СписокИсключений - это разделенный запятыми список типов исключений, которые метод может выбрасывать.

```
class ThrowsDemo {
   static  void   throwOne( )   throws   IllegalAccessException{
                System.out.println("Внутри throwOne.");
                throw new IllegalAccessException("demo");    }
  public static void main(String args[]){
      try {
          throwOne();   }
     catch (IllegalAccessException e) {
           System.out.println("Перехвачено" + e);    }
  }  }
```

Метод **public String toString()** возвращает строку, содержащую описание объекта и вызывается автоматически, когда объект выводится с помощью метода **println().**

finally

finally создает блок кода, который всегда будет выполнен после завершения блока **try{}catch{}**, перед кодом, следующим за **try{}catch{}**.

Блок **finally** выполняется независимо от того, возбуждено исключение или нет. Если исключение возбуждено, блок **finally** выполняется, даже если ни один оператор **catch** этому исключению не соответствует.

Это нужно для закрытия файлов, либо освобождения ресурсов, которые были получены в начале работы текущего метода и должны быть освобождены перед возвратом.

Оператор **finally** необязателен, но каждый оператор **try** требует наличия, по крайней мере, одного оператора **catch** или **finally**.

finally

```
class  FinallyDemo {

        static void procA( ) {
            try {
                System.out.println(“внутри procA”);
                throw  new  RuntimeException (“demo”) ;  }
            finally {
                System.out.println("блок finally procA");   }    }

        static void procB() {
            try {
                  System.out.println("Внутри procB");
                   return;   }
            finally {
                System.out.println("блок  finally  procB"); }     }
```

finally

```
static void procC () {
     try {
            System.out.println("внутри procC");      }
    finally {
            System.out.println("блок finally procC");  }       }

public static void main(String args[]) {
        try {
             procA( ) ;   }
       catch (Exception e) {
            System.out.println("Исключение перехвачено");   }
             procB ( );
             procC( );                }       }
```

Результат работы программы:

внутри procA

блок finally procA

Исключение перехвачено

внутри procB

блок finally procB

внутри procC

блок finally procC

Некоторые методы, определенные в Throwable

String getMessage( ) - возвращает описание исключения.

void printStackTrace( ) - отображает трассировку стека.

public String toString( ) - возвращает объект String, содержащий описание исключения.
(Версия метода **toString()**, определенная **Throwable**, сначала отображает имя исключения, двоеточие, а после - описание. Переопределив **toString( )**, можно сформировать «свою» строку для описания ошибки.)

Создание собственных подклассов исключений

Иногда требуется создать собственные типы исключений для обработки ситуаций, специфичных для ваших приложений. Для этого необходимо определить подкласс **Exception** (который является подклассом **Throwable**).

Класс **Exception** не определяет никаких собственных методов. Он наследует методы, представленные в **Throwable**. Поэтому всем исключениям, включая «собственные», доступны методы, определенные в **Throwable.**

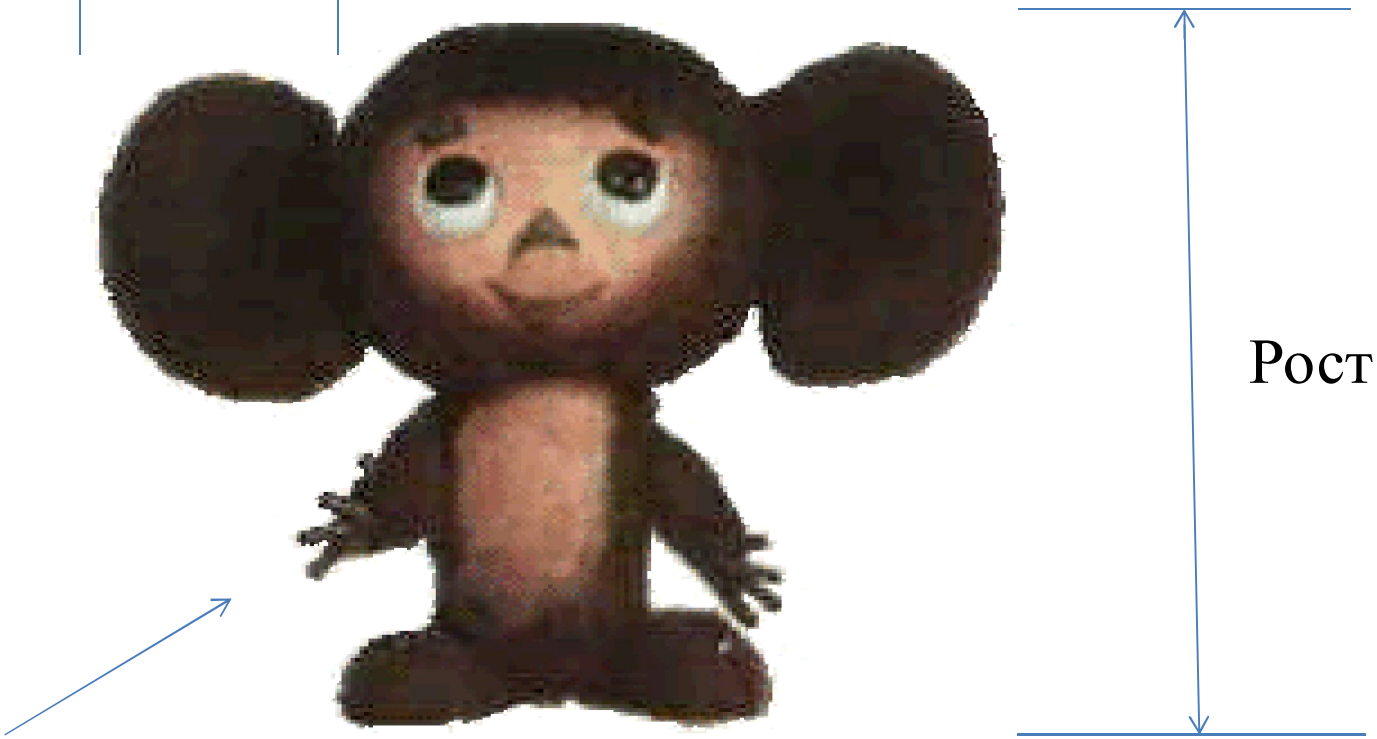
```
class MyException extends Exception {

    private int A;

    MyException( int   a ) {  A = a; }

    public String toString() {
         return  “Строка, описывающая исключение”;
                    }
  }
```

Диаметр уха



Рост

Цвет: Коричневый

```
import java.lang.Math;
import java.io.*;

class cheburashka{

   double Rost;
   double DiametrUha;
   String color;

  cheburashka(){
      Rost = 50;
      DiametrUha = 25;
      color = “Коричневый”;    }

  cheburashka(double R, double DU, String C) throws MyException{
      if (R<=0) throw new MyException(1);
      if (DU<=0) throw new MyException(2);
            Rost = R;
            DiametrUha = DU;
            color = C;   }
```

```
 double PerimetrUhaCheburashki(){
                              return Math.PI*DiametrUha;   }

  void  info(){
   System.out.println(“\n Чебурашка: \n Цвет: " + color + "\n Рост: "
                    + Rost + "\n Диаметр уха: " + DiametrUha +
                "\n Периметр уха: " + PerimetrUhaCheburashki());    }
}
class MyException extends Exception {
   private int A;

   MyException(int a){  A = a; }

   public String toString() {
     if(A==1) return “Рост чебурашки не может быть меньше 0!”;
     if(A==2) return “Ну ооочень маленькие уши у чебурашки!”;
     return “ “;  }
}
```

```
class CheburashkiSobrabotkoi{

  public static void main(String args[]) throws IOException{
     double Rost=0, DiametrUha=0;
     String color;
     String StrokaVvoda;
     int f=0;
BufferedReader bf = new BufferedReader(new InputStreamReader(System.in));

  cheburashka CH1 = new cheburashka();
  CH1.info();
```

```
while(f==0){
  try{
    StrokaVvoda = bf.readLine();
    Rost = Double.parseDouble(StrokaVvoda);
    StrokaVvoda = bf.readLine();
    DiametrUha = Double.parseDouble(StrokaVvoda);
    color = bf.readLine();
    cheburashka CH2 = new cheburashka(Rost, DiametrUha, color);
    f=1;
    CH2.info();       }
  catch(NumberFormatException e){
    System.out.println("Неверный формат числа! ");
            continue;     }
  catch(MyException e){
            System.out.println(e);
            continue;
              }
   } } }
```