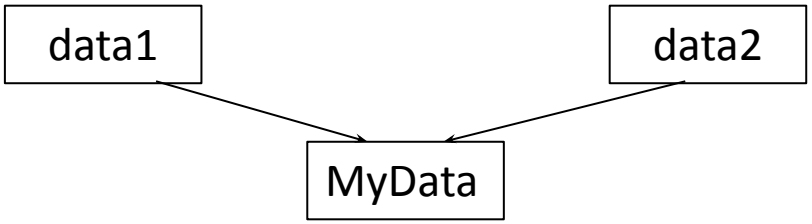
Дублирование ссылок и Клонирование объектов

При обыкновенном присваивание объектов передаются ссылки на объект. В итоге два экземпляра ссылаются на один объект, и изменение одного приведет к изменению другого.

MyData data1 = new(MyData); MyData data2 = data1;



Клонирование – порождение нового объекта независимого от существующего.

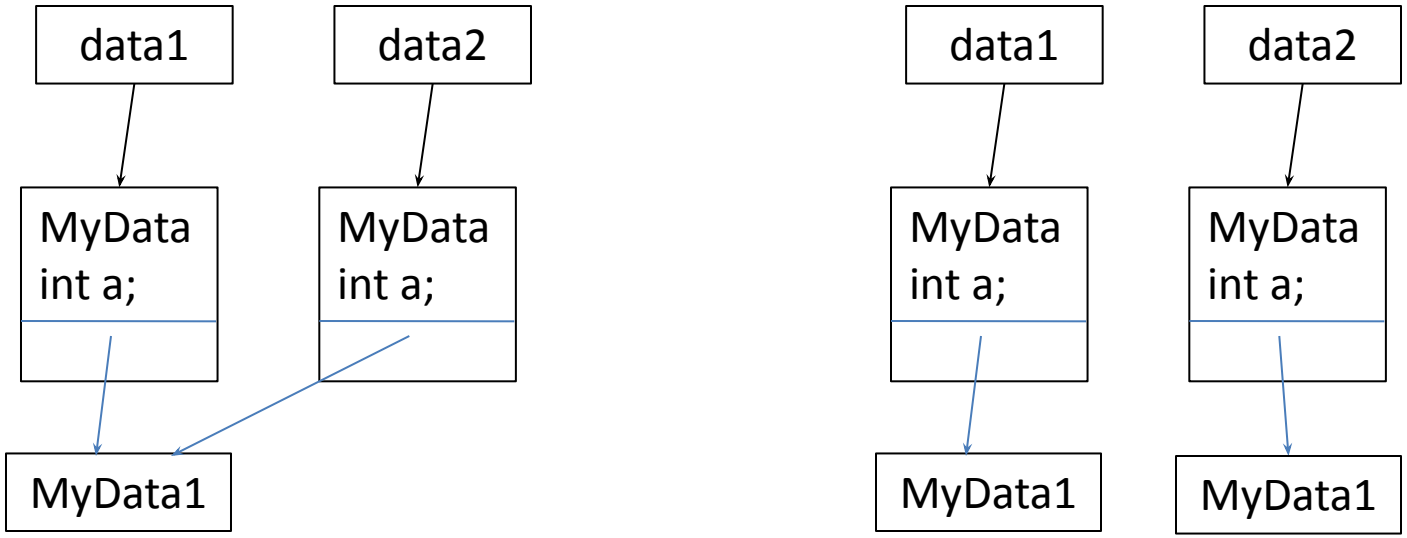
MyData data2 = data1.clone();

data2 ≠ data1

Типы клонирования

Клонирование объекта бывает 2-х видов:

- Поверхностное (shallow); в этом случае копируются значения простых полей и ссылочные значения.
- Глубокое (deep); клонирование составного объекта, это достигается через рекурсивное клонирование составляющих его объектов.



Переопределение метода clone

В *Java* у базового класса *java.lang.Object* существует метод **clone**, с помощью которого можно создать новый объект точно такой же как и текущий.

Метод *clone* в классе *Object* является защищенным (*protected*)

protected native Object clone() throws CloneNotSupportedException;

native, говорит о том, что его реализация предоставляется виртуальной машиной.

Чтобы этот метод сделать общедоступным следует его переопределить в подклассе с областью видимости *public и* реализовывать интерфейс Cloneable.

Интерфейс Cloneable не реализует ни одного метода. Он является всего лишь маркером, говорящим, что данный класс реализует клонирование объекта.

Если класс не реализует данный интерфейс, то будет выброшено исключение *CloneNotSupportedException*.

```
public class MyData implements Cloneable {
        public int a;
       MyData1 d;
public MyData clone() throws CloneNotSupportedException {
        MyData obj = (MyData)super.clone(); // поверхностное копирование
```

Глубокое клонирование

```
public class MyData implements Cloneable {
        public int a;
       MyData1 d;
public MyData clone() throws CloneNotSupportedException {
        MyData obj = (MyData)super.clone(); // поверхностное копирование
       if (this.d != null)
      obj.d = this.d.clone(); // глубокое копирование
        return obj;
    }
  }
```

При клонировании в глубину нельзя использовать конструктор:

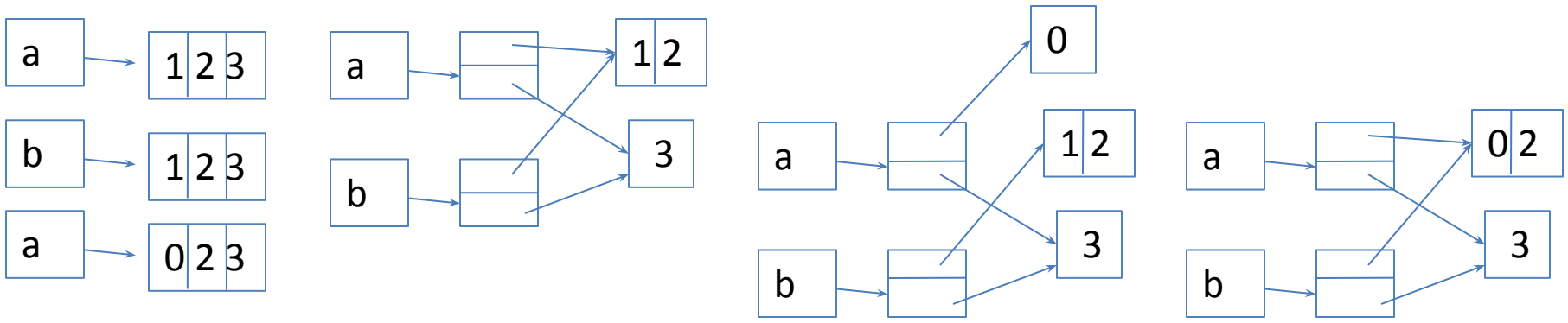
obj.d= new MyData1();

Поле d может содержать ссылку на базовый класс **MyData1**

Клонирование массивов

```
int a[]={1, 2, 3}; // один объект
int b[]=(int[])a.clone(); // поверхностное копирование
a[0]=0;
System.out.println(b[0]); // 1

int a[][]={{1, 2}, {3}}; // три объекта
int b[][]=(int[][]) a.clone();
{ a[0]=new int[]{0}; System.out.println(b[0][0]); }  // первый вариант: 1
{ a[0][0]=0; System.out.println(b[0][0]); }; // второй вариант: 0
}
```



Понятие эквивалентности

Метод **equals()** обозначает отношение эквивалентности объектов. Эквивалентным называется отношение, которое является симметричным, транзитивным, рефлексивным и постоянным.

Рефлексивность: для любого ненулевого x, x.equals(x) вернет true;

Транзитивность: для любого ненулевого x, y и z, если x.equals(y) и y.eqals(z) вернет true, тогда и x.equals(z) вернет true;

Постоянство: для любых объектов *x* и *y* x.equals(y) возвращает одно и тоже, если информация, используемая в сравнениях, не меняется;

Симметричность: для любого ненулевого x и y, x.equals(y) должно вернуть true, тогда и только тогда, когда y.equals(x) вернет true**.**

Реализация equals в классе Object

Для определения равенства различных объектов применяется метод **equals**. Метод **equals** реализован в классе Object и соответственно наследуем любым классом Java. В базовом классе Object метод equals сравнивает содержимое объектов.

public boolean equals(Object obj) { return (this == obj); }

При сравнение объектов, операция “==” вернет true лишь в одном случае — когда ссылки указывают на один и тот же объект. В данном случае не учитывается содержимое полей.

```
class A {
String objectName;
A (String name) { objectName = name; }      // Конструктор
}
public  class MyA {
public  static  void main (String args[ ])
{
A  A_1 = new A(«Строка1");      // Создание экземпляра класса
A  A_eq = A_1;       // Ссылка на существующий объект
A  A_clon = (A)A_1.clone;      // Создание объекта методом clone
A  A_2 = new A(«Строка1");
 // Сравнение объектов:
if (A_1.equals(A_eq))    { истина }
if (A_1.equals(A_ clon)) { ложь }
if (A_1.equals(A_ 2))    {  ложь}
} }
```

Переопределение метода equals

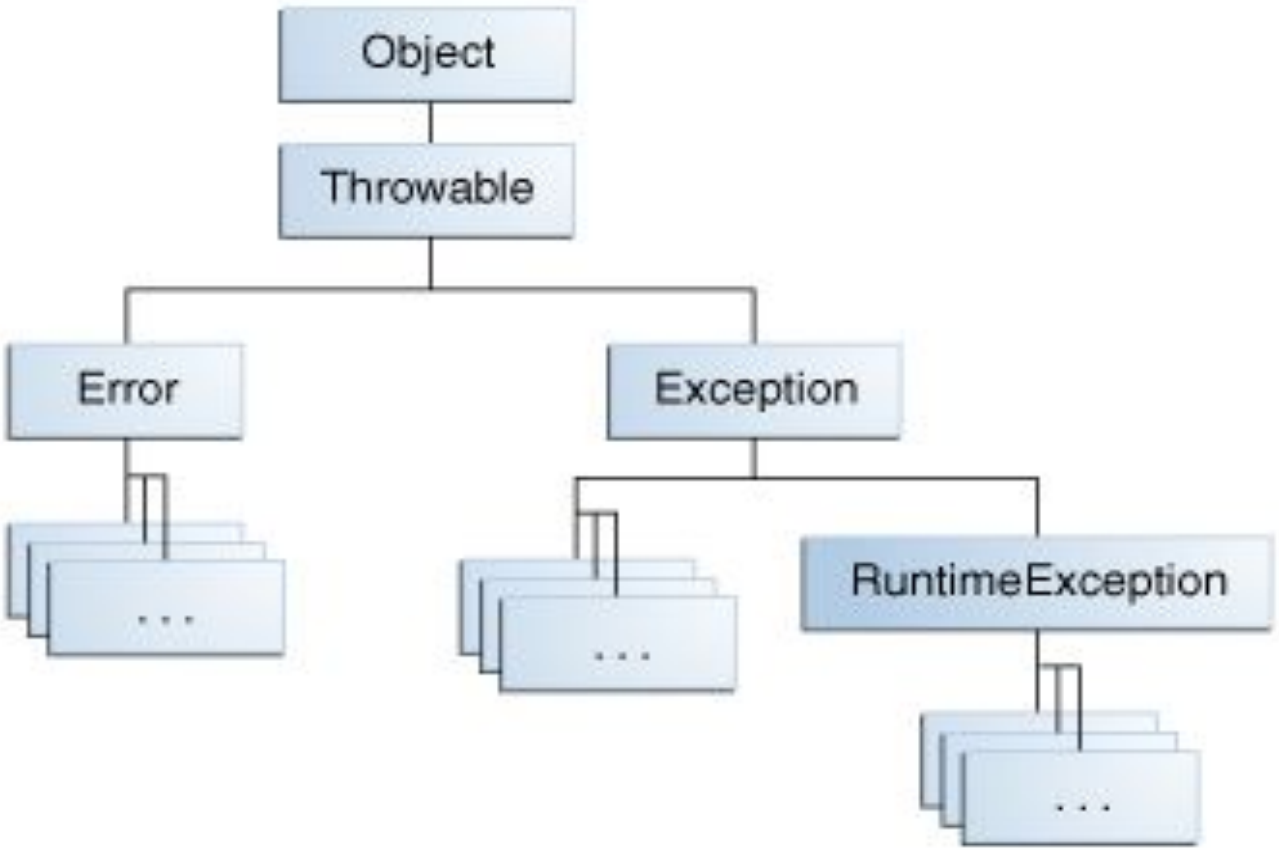
```
public class App
{  String str1 = new String("Test1");       String str2 = new String("Test2");
     int num = 5;
     public boolean equals(Object obj) {
         if(obj == null) {return false;} // проверяет не равен ли obj – null
   if(!(obj instanceof App)){return false;} // проверяет является ли obj объектом
     App
         App obj1 = (App) obj;
          // сравнивает поля экземпляров класса
          return str1.equals(obj1.str1) && str2.equals(obj1.str2) && num == obj1.num;
     }
        public static void main( String[] args )
   {          App app1 = new App();
      App app2 = new App();
          System.out.println( app1.equals(app2) ); // результат:  true
   }
}
```

Причины возникновения исключений

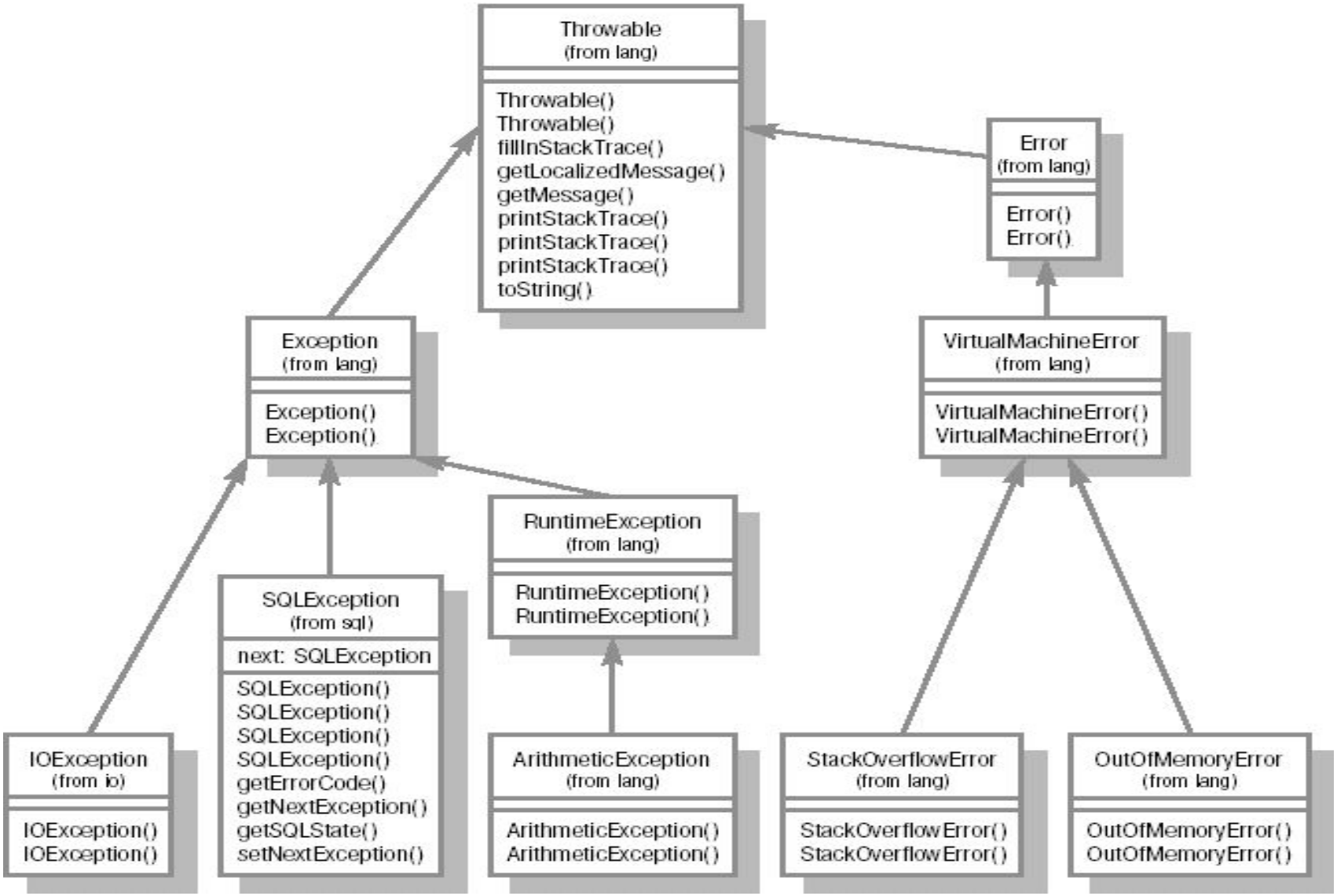
- Попытка выполнить некорректное выражение. Например, деление на ноль, или обращение к объекту по ссылке, равной null, попытка использовать класс, описание которого отсутствует, и т.д.
- Выполнение оператора throw Этот оператор применяется для явного порождения исключения.
- Асинхронные ошибки во время исполнения программы. Причиной таких ошибок могут быть сбои внутри самой виртуальной машины

Исключения (Exceptions)

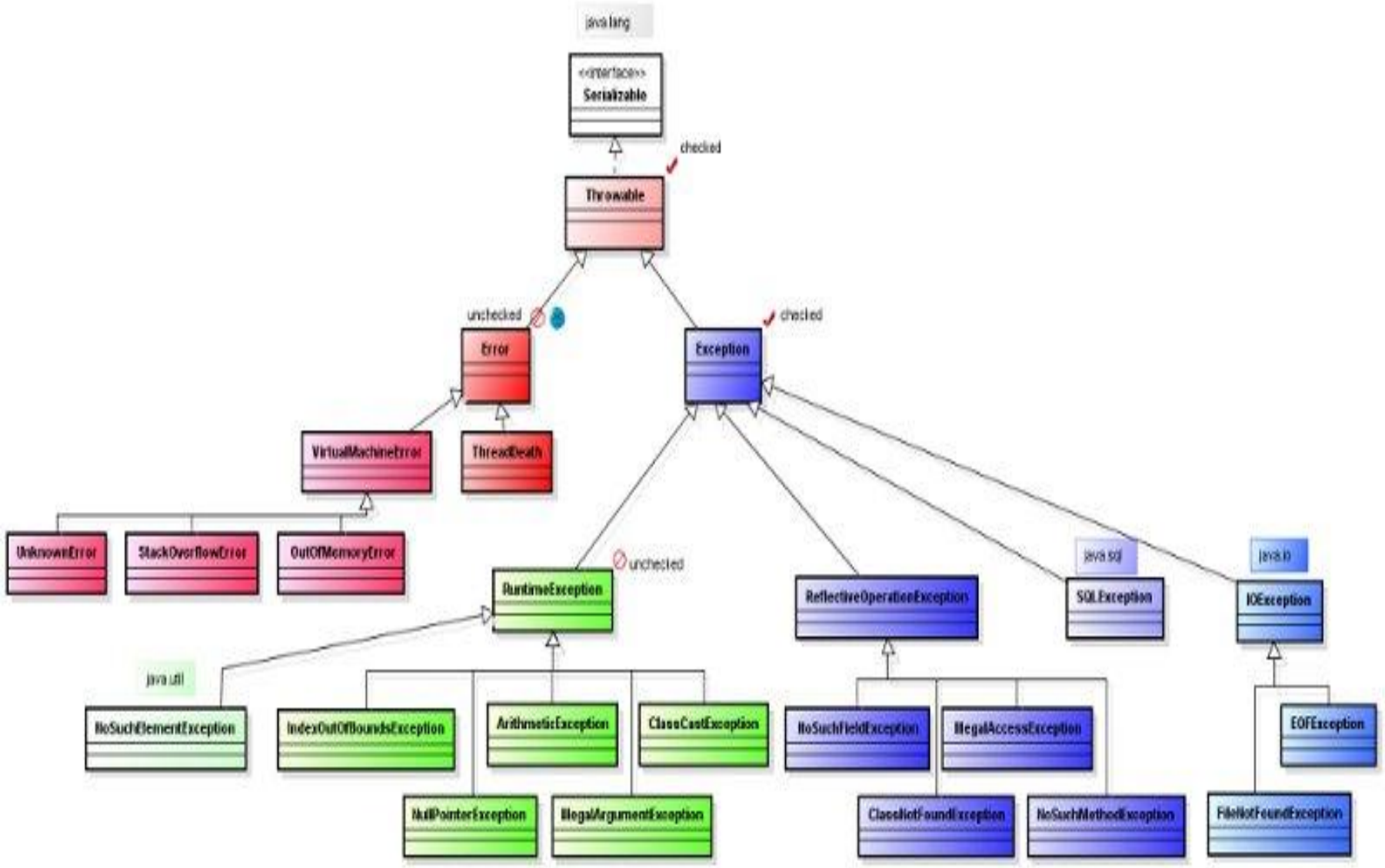
Исключениями или исключительными ситуациями (состояниями) называются ошибки, возникшие в программе во время её работы.



Иерархия классов стандартных исключений



Иерархия классов стандартных исключений



Основные методы базового типа Throwable

Throwable fillInStackTrace() - возвращает объект класса **Throwable**, содержащий полную трассировку стека.

String getLocalizedMessage() - возвращает локализованное описание исключения

String getMessage() - возвращает описание исключения

void printStackTrace() - отображает трассировку стека

void printStackTrace(PrintStream stream) - посылает трассировку стека в заданный поток

void printStackTrace(PrintWriter stream) - посылает трассировку стека в заданный поток

String toString() - возвращает объект класса **String**, содержащий описание исключения.

Проверяемые и непроверяемые исключения

Все исключения, порождаемые от Throwable, можно разбить на три группы. Они определяются тремя базовыми типами: наследниками Throwable - классами Error и Exception, а также наследником Exception - RuntimeException.

- Ошибки, порожденные от Exception (и не являющиеся наследниками RuntimeException), являются проверяемыми. Т.е. во время компиляции проверяется, предусмотрена ли обработка возможных исключительных ситуаций. Как правило, это ошибки, связанные с окружением программы (сетевым, файловым вводом-выводом и др.), которые могут возникнуть вне зависимости от того, корректно написан код или нет. Таким образом повышается надежность программы, ее устойчивость при возможных сбоях.
- Исключения, порожденные от RuntimeException, являются непроверяемыми и компилятор не требует обязательной их обработки. Как правило, это ошибки программы, которые при правильном кодировании возникать не должны (например, IndexOutOfBoundsException - выход за границы массива, java.lang.ArithmeticException - деление на ноль). Поэтому, чтобы не загромождать программу, компилятор оставляет на усмотрение программиста обработку таких исключений с помощью блоков try-catch.
- Исключения, порожденные от Error, также не являются проверяемыми. Они предназначены для того, чтобы уведомить приложение о возникновении фатальной ситуации, которую программным способом устранить практически невозможно (хотя формально обработчик допускается). Они могут свидетельствовать об ошибках программы, но, как правило, это неустранимые проблемы на уровне JVM. В качестве примера можно привести StackOverflowError (переполнение стека), OutOfMemoryError (нехватка памяти).

Неперехваченные исключения

```
package etu.lab.exp;
classExc 0 {
    public static void main(string args[]) {
    int d = 0;
    int a = 42 / d;
    }
    }
```

Exception in thread "main" java.lang.ArithmeticException: / by zero

at etu.lab.exp.Exc0.main(Exc0.java:5)

```
package etu.lab.exp;
class Exc0 {
    static void subroutine() {
    int d = 0;
    int a= 10 / d;
    }
    public static void main(String args[]) {
    Exc0 .subroutine();
    }
    }
```

Exception in thread "main" java.lang.ArithmeticException: / by zero

at etu.lab.exp.Exc0.subroutine(Exc0.java:5) at etu.lab.exp.Exc0.main(Exc0.java:8)

Использование оператора throw и throws

Оператор throw используется для возбуждения исключения «вручную».

```
public int calculate(int theValue)
{ if( theValue < 0) { throw new Exception( "Параметр для вычисления не должен быть отрицательным");
// Обработка исключительной ситуации
} }
```

Если метод способен возбуждать исключения, которые он сам не обрабатывает, он должен объявить о таком поведении, чтобы вызывающие методы могли защитить себя от этих исключений.

```
public int calculate(int theValue) throws Exception
{ if( theValue < 0) { throw new Exception( " Параметр для вычисления не должен быть отрицательным "); } }
```

Конструкция try-catch

```
try { // Код, который может сгенерировать исключение }
catch(Type1 id1) { // Обработка исключения Type1 }
catch(Type2 id2) { // Обработка исключения Type2 }
catch(Type3 id3) { // Обработка исключения Type3 }
// продолжение программы
```

```
try { ... }
catch(IOException ex) {
 ...
// Обработка исключительной ситуации
 ...
 // Повторное возбуждение исключительной  ситуации
throw ex; }
```

Конструкция try-catch-finally

```
try { // Критическая область, при которой могут быть выброшены A, B или C }
catch(A a1) { // Обработчик ситуации A }
catch(B b1) { // Обработчик ситуации B }
catch(C c1) { // Обработчик ситуации C }
finally { // Действия, совершаемые всякий раз }
int getNumber(){
 try { //Исключения есть или нет
return 0;
}
catch(Exception e){
return 1;
 }
 finally {
return 2;
}
return 3;
}
```

Ответ: 2.

Пример обработки исключения

```
class ThrowDemo {
    static void demoproc() {
    {
            throw new NullPointerException("demo");
    }
    catch (NullPointerException e) {
            System.out.println(“обработка внутри demoproc");
             throw e;  // повторный вызов исключения
    }
    }
    public static void main(String args[]) {
    try {
              demoproc();
    }
    catch(NulPointerException e) {
               System.out.println(“повторно:" + e);
    }
    }
    }
```

Создание пользовательских классов исключений

```
class FactorialException extends Exception{
    private int number;
   public int getNumber(){return number;}
   public FactorialException(String message, int num){
          super(message); // в базовый класс передаем сообщение
      number=num;
   }
}
class Factorial{
    public static int getFactorial(int num) throws FactorialException{
          int result=1;
      if(num<1) throw new FactorialException("Число не может быть меньше 1", num);
            for(int i=1; i<=num;i++){ result*=i;
      }
      return result;
   }
}

public static void main(String[] args){
          try{
      int result = Factorial.getFactorial(6);
      System.out.println(result);
   }
   catch(FactorialException ex){
      System.out.println(ex.getMessage()); // вывод сообщения
      System.out.println(ex.getNumber());
   }
}
```

Переопределение методов и исключения

```
public class BaseClass{
public void method () throws IOException { ... }
 }
public class LegalOne extends BaseClass {
public void method () throws IOException { ... } // корректно (список ошибок не изменился);
 }
public class LegalTwo extends BaseClass {
public void method () { ... }   // корректно (новый метод не может выбрасывать ошибок)
}
public class LegalThree extends BaseClass {
public void method () throws EOFException { ... }  // корректно
// (новый метод может создавать  исключения, которые являются подклассами
      исключения)
}
public class IllegalOne extends BaseClass {
public void method () throws IOException, IllegalAccessException { ... } // некорректно
// ( IllegalAccessException не является подклассом IOException, список расширился);
}
public class IllegalTwo extends BaseClass {
public void method () { ... throw new Exception(); }         } // некорректно
//(в теле метода бросается исключение, не указанное в throws)
 }
```

Основные правила обработки исключений

- обработать ошибку на текущем уровне (избегайте перехватывать исключения, если не знаете, как с ними поступить)
- исправить проблему и снова вызвать метод, возбудивший исключение
- предпринять все необходимые действия и продолжить выполнение без повторного вызова действия
- сделать все возможное в текущем контексте и заново возбудить это же исключение, перенаправив его на более высокий уровень
- сделать все, что можно в текущем контексте, и возбудить новое исключение, перенаправив его на более высокий уровень
- завершить работу программы