Лекция 4

Оператор if-else

Оператор if – else имеет вид

if(условие) {…..}

else {…..}

Данный оператор полностью аналогичен условному оператору в C++.

Оператор switch

Оператор switch имеет вид

switch(целочисленное выражение или enum){

case метка1: оператор1,оператор2,…,break;

case метка2: оператор1,оператор2,…,break;

case метка3: оператор1,оператор2,…,break;

…………………………………

default:…………………

}

Данный оператор полностью аналогичен условному оператору в C++.

В Java 7 в switch можно использовать класс String

Пример:

```
public String getTypeOfDayWithSwitchStatement(String  dayOfWeekArg) {
  String typeOfDay;
  switch (dayOfWeekArg) {
    case "Monday": typeOfDay = "Start of work week"; break;
    case "Tuesday": case "Wednesday": case "Thursday":
                                                  typeOfDay = "Midweek"; break;
    case "Friday": typeOfDay = "End of work week"; break;
    case "Saturday": case "Sunday": typeOfDay = "Weekend"; break;
   default: throw  new IllegalArgumentException("Invalid day of
                                                    the week: " +
dayOfWeekArg);
  }
  return typeOfDay;
}
```

Циклы в Java.

1. Цикл while.

Цикл while имеет вид

while (условие) {…..};

данный цикл аналогичен циклу while в C++.

2. Цикл do – while.

Цикл do - while имеет вид

do{….} while (условие);

данный цикл аналогичен циклу while в C++.

3. Цикл for

Цикл for имеет вид

for(инициализация; условие; приращение){ …………………..

}

Пример:

```
For(int i=0,j=0;i<10;i++,j++){
 выражение
}
```

4. Цикл for-each.

Цикл for-each развитие цикла for(идея взята из языка Python). Пример1:

```
int[] a={1,2,3,4};
int sum=0;
for(int value:a){
    sum+=value;
}
```

Этот код эквивалентен коду:

```
int[] a={1,2,3,4};
int sum=0;
int value=0;
for(int i=0;i<a.length;i++){
   value=a[i];
   sum+=value;
}
```

Пример 2:

```
ArrayList a=new ArrayList(10);
for(int i=0;i<10;i++){
   Integer n=new Integer(i);
   a.add(i,n);}
int sum=0;
for(Object value:a){
      sum+=((Integer)value).intValue();
}
```

Таким образом, базовая структура цикла for-each имеет вид:

```
for (declaration : expression)
  statement
```

declaration – это переменная, которая должна иметь тип, совместимый с каждым элементом списка, массива или коллекции, по которым производится итерация.

expression - это выражение. Оно должно вычислять что-либо, по чему можно делать итерацию.

Результатом данного выражения может быть массив или тип класса, реализующего интерфейс Collection.

Оператор break

Оператор break применяется для выхода из любого блока.

Существует разновидность оператора break с меткой. Рассмотрим пример:

```
private float[][] Matix;
  public boolean workOnFlag(float flag) {
    int y, x;
    boolean found = false;
    search: //search это метка
    for (y = 0; y < Matrix.length; y++) {
      for (x = 0; x < Matrix[y].length; x++) {
        if (Matrix[y][x] == flag) {
          found = true;
          break search; //оператор break с меткой search
        }
      }
    }.
    if (!found) return false;
    return true;
}
```

Оператор continue

Оператор continue осуществляет переход в конец тела цикла и вычисляет значение управляющего логического выражения.

Этот оператор часто используется для пропуска некоторых значений в диапазоне цикла, которые должны игнорироваться.

```
int i=0;
while (i<10) {
……………………….
 if (i==5) continue;
 //в случае если условие i==5 истина, операторы ниже выполняться не будут
}
```

Существует оператор continue с меткой.

Рассмотрим пример использования continue

```
public class MyTest {
  public static void main(String[] args) {
 label1: for (int i = 0; i < 3; i++) {
          if (i == 1)  continue label1;
            System.out.print("TEST ");
        }
   }
}
```

На экране получим TEST TEST

Оператор return

Оператор return завершает выполнение метода и передает управление в точку его вызова.

Если метод не возвращает никакого значения, достаточно написать:

return;

Оператор goto в Java отсуствует.

Но слово goto является зарезервированным.

Исключения

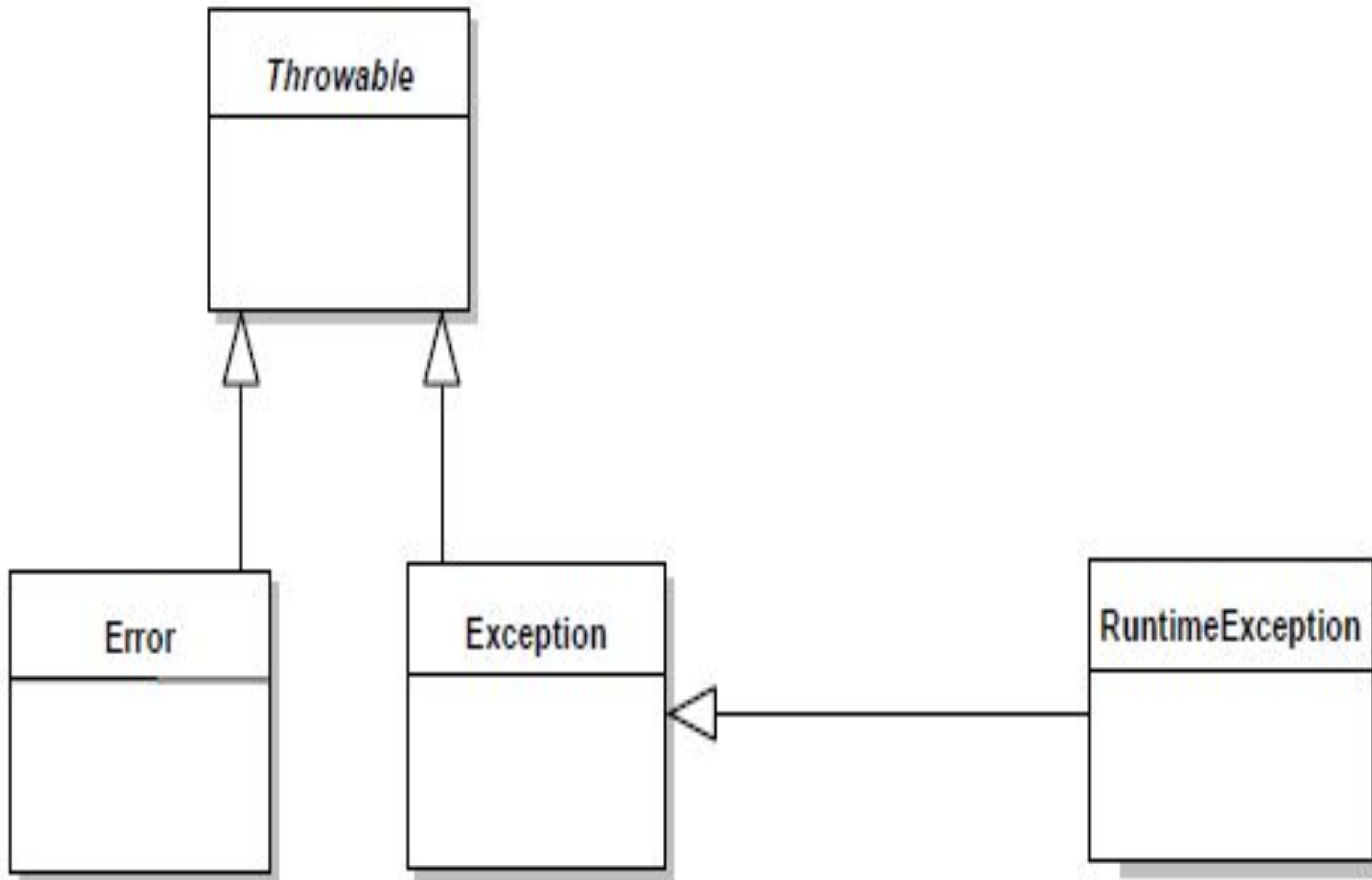
Исключения в Java представляют собой объекты.

Все типы исключений (то есть все классы, объекты которых возбуждаются в качестве исключений) должны расширять класс языка Java, который называется Throwable, или один из его подклассов.

Однако, по соглашению, новые типы исключений расширяют класс Exception, который является наследником Throwable.

Другая ветвь дерева подклассов Throwable — класс Error, который предназначен для описания исключительных ситуаций, которые при обычных условиях не должны перехватываться в пользовательской программе.

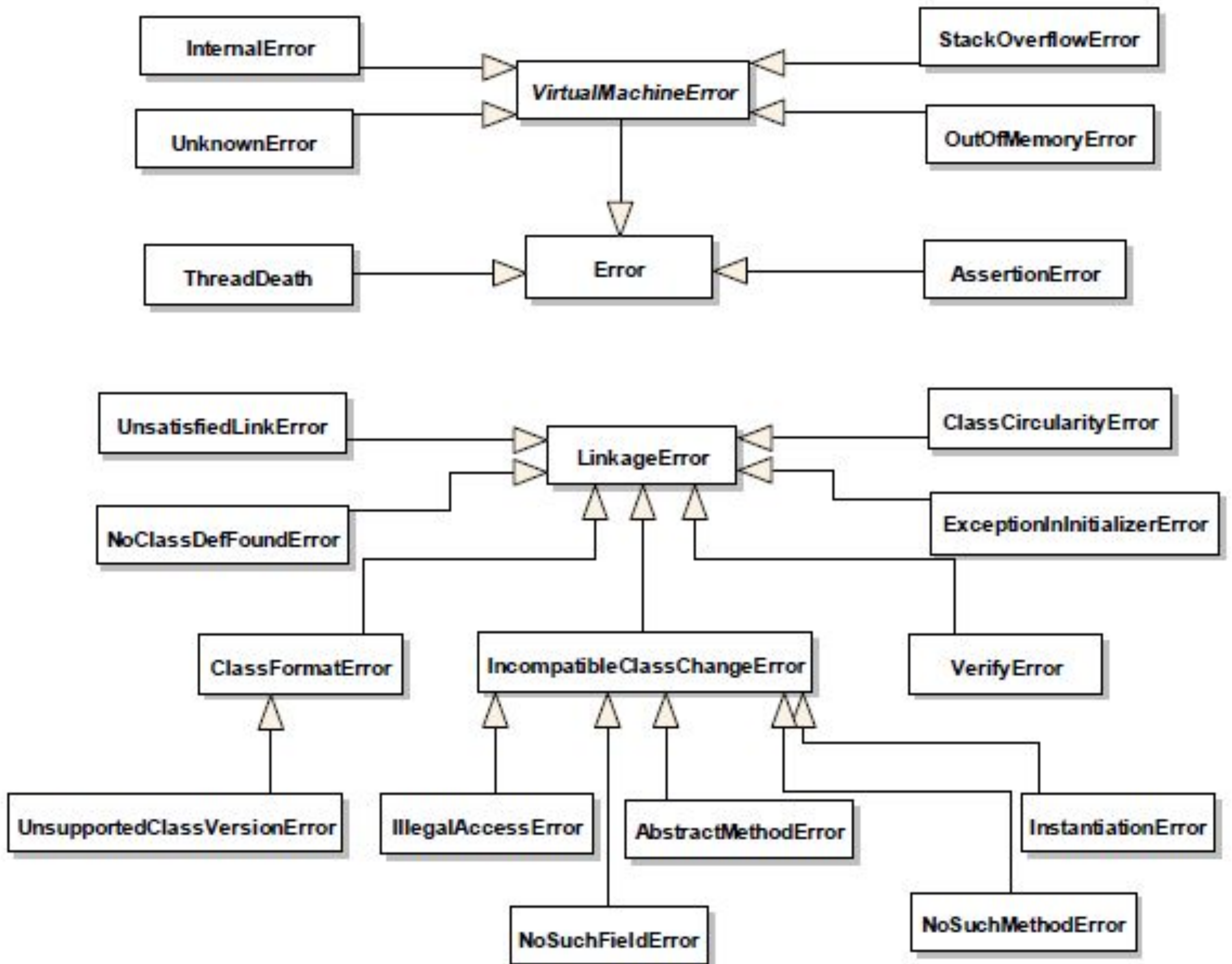
Таким образом иерархия исключений имеет вид:



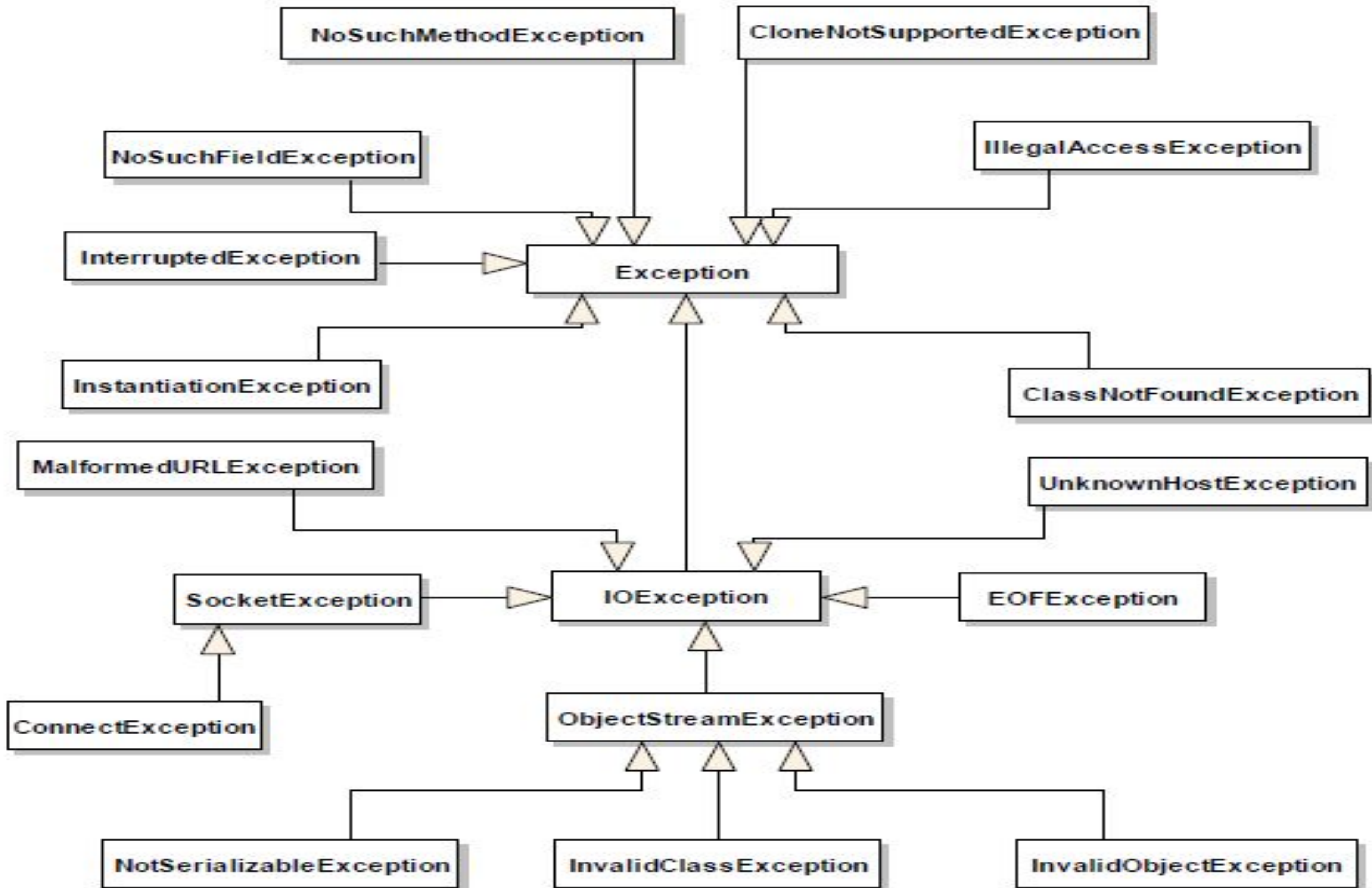
Исключительные ситуации типа **Error** возникают только во время выполнения программы.

Такие исключения связаны с серьезными ошибками, к примеру – переполнение стека, и не подлежат исправлению и не могут обрабатываться приложением.

Иерархия классов, наследуемых от класса **Error** имеет вид


InternalError
VirtualMachineError
StackOverflowError
UnknownError
OutOfMemoryError
ThreadDeath
Error
AssertionError
UnsatisfiedLinkError
LinkageError
ClassCircularityError
NoClassDefFoundError
ExceptionInInitializerError
ClassFormatError
IncompatibleClassChangeError
VerifyError
UnsupportedClassVersionError
IllegalAccessError
AbstractMethodError
InstantiationError
NoSuchFieldError
NoSuchMethodError

Иерархия классов исключений, наследуемых от класса **Exception** имеет вид:



Проверяемые исключения должны быть обработаны в методе, который может их генерировать, или включены в **throws**-список метода для дальнейшей обработки в вызывающих методах.

Возможность возникновения проверяемого исключения может быть отслежена на этапе компиляции кода.

Рассмотрим пример проверяемого пользовательского исключения:

```
public class ListFull extends Exception {
  private String attrName;
  public ListFull(String n){ attrName=n;}
  public String getMessage(){
    return attrName;}
}
```

Во время выполнения могут генерироваться также исключения, которые могут быть обработаны без ущерба для выполнения программы.

В отличие от проверяемых исключений, класс **RuntimeException** и порожденные от него классы относятся к непроверяемым исключениям. Компилятор не проверяет, генерирует ли и обрабатывает ли метод эти исключения.

Исключения типа **RuntimeException** автоматически генерируются при возникновении ошибок во время выполнения приложения

Таким образом, нет необходимости в проверке генерации исключения вида:

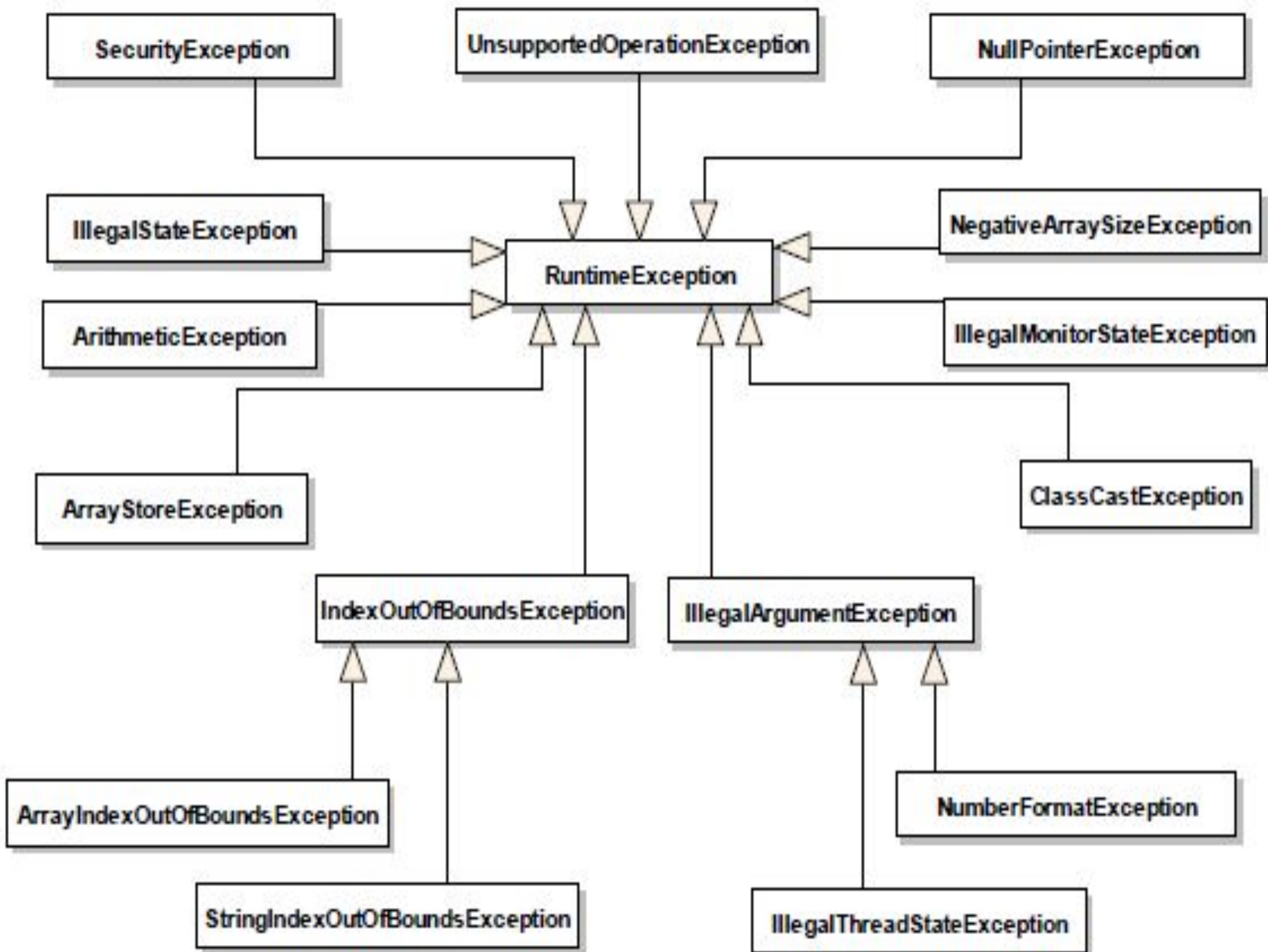
```
if(a==null) throw new NullPointerException();
```

объект класса **NullPointerException** при возникновении ошибки будет сгенерирован автоматически.

Кроме этого, в любом случае нет необходимости в обработке этого исключения непосредственно в методе или в передаче на обработку вызывающему методу с помощью оператора **throw**.

В итоге исключение будет передано в метод **main()**, где обрабатывается вызовом метода **printStackTrace()**, выдающего данные трассировки.

Иерархия этих исключений имеет вид:


SecurityException
UnsupportedOperationException
NullPointerException
IllegalStateException
RuntimeException
NegativeArraySizeException
ArithmeticException
IllegalMonitorStateException
ArrayStoreException
ClassCastException
IndexOutOfBoundsException
IllegalArgumentException
ArrayIndexOutOfBoundsException
NumberFormatException
StringIndexOutOfBoundsException
IllegalThreadStateException

Оператор throw

Исключение генерируется инструкцией throw

throw выражение;

Рассмотрим пример:

```
class List{
  Node first;
  int size;
  int max;
  class Node{
     int value;
     Node next;
  }
  ……………
 public void add(int el) throws ListFull {
   …………..
   if(size>max) throw new ListFull("List is full");
     ………………
 }
};
```

Условие throws.

Проверяемые исключения, выбрасываемые методом, объявляются в условии throws, которое может содержать список значений, отделяемых друг от друга запятыми:

```
public void f() throws Exception1,Exception2 {
   ......
};
```

При переопределении унаследованного метода или реализации метода абстрактного класса необходимо обеспечить совместимость предложения метода подвергающегося переопределению.

Существует следующее правило – при переопределении или реализации не позволяется задавать в предложении throws нового метода больше объявляемых исключений, нежели в исходном коде.

Если в объявлении метода стоит native, то в этом методе тоже можно объявлять исключения, однако реализация таких native – методов находится вне компетенции компилятора Java.

Операторы try, catch и finally

Чтобы перехватить исключение, необходимо поместить фрагмент программы в оператор try. Базовый синтаксис оператора try выглядит следующим образом:

```
 try{
  ……………………..
 }
catch (тип-исключения идентификатор)
{……}
catch (тип-исключения идентификатор)
{……}
finally {……}
```

Рассмотрим пример:

```
 public void f(){
  List l=new List(…);
  ………………
try{
 …………..
 l.add(10);
 ……………
}
catch(List_full e){
  System.out.println(e.getMessage());
}
 ………………………………..
}
```

Необходимо помнить, что **свойством транзакционности исключения не обладают – действия, произведенные в блоке try *до* возникновения исключения, не отменяются *после* его возникновения.**

Рассмотрим пример. Утечка памяти в Java

```
public class PartialInitTest{
  static PartialInitTest self;
  private int field1 = 0;
  private int field2 = 0;
  public PartialInitTest(boolean fail) throws Exception {
    self = this;
    field1 = 1;
    if (fail) { throw new Exception(); }
    field2 = 1;
  }
```

```
 public boolean isConsistent(){
   return field1 == field2;
 }
 public static void main(String[] args){
   PartialInitTest pit = null;
   try {
      pit = new PartialInitTest(true);
   } catch (Exception ex){ // do nothing }
   System.out.println("pit: "+pit);
   System.out.println("PartialInitTest.self reference:”
                                                          +PartialInitTest.self);
   System.out.println("PartialInitTest.self.isConsistent():“
                                  +PartialInitTest.self.isConsistent());
 }
}
```

На экране получим:

pit: null

PartialInitTest.self reference:

test.PartialInitTest@1e0bc08

PartialInitTest.self.isConsistent(): false

Предложение finally

Предложение finally оператора try позволяет выполнить некоторый фрагмент программы независимо от того, произошло исключение или нет.

Обычно работа такого фрагмента сводится к “чистке” внутреннего состояния объекта или освобождению “необъектных” ресурсов (например, открытых файлов), хранящихся в локальных переменных.

Рассмотрим пример:

```
public boolean searchFor(String file, String word) throws
                                                   StreamException{
  Stream input = null;
  try {
    input = new Stream(file);
    while (!input.eof())
       if (input.next() == word) return true;
    return false;}
  finally {
   if (input != null) input.close();
 }}
```

Если создание объекта оператором new закончится неудачно, то input сохранит свое исходное значение null.

Если же выполнение new будет успешным, то input будет содержать ссылку на объект, соответствующий открытому файлу.

Во время выполнения условия finally поток input будет закрываться лишь в том случае, если он предварительно был открыт.

Независимо от того, возникло ли исключение при работе с потоком или нет, условие finally обеспечивает закрытие файла.

Таким образом, общая форма блока обработки исключений имеет вид:

```
try {
  // блок кода
}
catch (ТипИсключения1 e) {
   // обработчик исключений типа ТипИсключения1
 }
catch (ТипИсключения2 e) {
   // обработчик исключений типа ТипИсключения2
    throw(e)   // повторное возбуждение исключения
}
finally {
}
```

Повторное возбуждение исключения

В Java возможно повторное возбуждение исключения.

Рассмотрим пример:

```
class MyExcep extends Exception {…………..};
class B{
   public static void f(int k) throws MyExcep{
     try{
       if (k>0) throw new MyExcep();
     }catch(MyExcep obj){
       if (k==3) throw (obj);
     }
  }
};
```

```
public class Main{
   public static void main(String[] args) {
       B pb=new B();
       try{
         pb.f(0);
       }catch(MyExcep ob){………………….. };
   }
}
```

Отладочный механизм assertion

На этапе отладки найти неявные ошибки в функционировании приложения бывает довольно сложно.

Определять и исправлять такие ситуации позволяет механизм проверочных утверждений (assertion).

При помощи этого механизма можно сформулировать требования к входным, выходным и промежуточным данным методов классов в виде некоторых логических условий.

Рассмотрим пример:

Например, попытка обработать ситуацию появления отрицательного возраста может выглядеть следующим образом:

```
int age = ob.getAge();
if (age >= 0) {
  // реализация
 } else {
  // сообщение о неправильных данных
}
```

Механизм assertion позволяет создать код, который будет генерировать исключение на этапе отладки проверки постусловия или промежуточных данных в виде:

int age = ob.getAge();

assert (age >= 0): "NEGATIVE AGE!!!";

// реализация

Правописание инструкции **assert**:

assert (boolexp): expression;

assert (boolexp);

Выражение **boolexp** может принимать только значение типов **boolean** или **Boolean**, а **expression** – любое значение, которое может быть преобразовано к строке.

Если логическое выражение получает значение **false**, то генерируется исключение **AssertionError**, и выполнение программы прекращается с выводом на консоль значения выражения **expression** (если оно задано).

Assertion можно включать для отдельных классов и пакетов при запуске виртуальной машины в виде:

java –enableassertions MyClass

или

java –ea MyClass

Для выключения применяется **–da** или

-disableassertions.

В Java 7 появились некоторые новые черты работы с исключениями.

Некоторые ресурсы, создаваемые в процессе работы приложения, должны быть закрыты явно - обычно с помощью метода close().

В Java 7 расширен функционал try блока, позволяя прямо в try блоке декларировать необходимые ресурсы, которые по завершению блока будут корректно закрыты (с помощью вызова close()).

Для того, чтобы среда могла определить ресурсы требующие и поддерживающие явное закрытие, был создан новый интерфейс **Closable** и соответсвующие классы ресурсов (InputStream, Writers, Sockets,

Sql классы) расширены для реализации этого интерфейса.

Такая реализация будет обратно совместима со старыми версиями Java.

Рассмотрим пример. Ранее необходимо было писать

```
BufferedReader br = new BufferedReader(
                    new FileReader(path));
try {
    return br.readLine();
} finally {
    br.close();
}
```

В Java 7 можно написать

```
try (BufferedReader br =
  new BufferedReader(new FileReader(path)) {
    return br.readLine(); }
```

Вложенные классы и интерфейсы

Статический вложенный класс

Вложенный класс или интерфейс, объявленный в виде статического члена внешнего класса ведет себя точно так же как и обычный внешний класс.

Имя вложенного типа задается в виде **ИмяВнешнегоТипа.ИмяВложенногоТипа**

Рассмотрим пример:

```
public class BankAccount  {
 private long number;
 private long balance;
 public static class Permissions  {
   public boolean canDeposit, canwithdraw,
                                   canclose;
}
…………}
```

Создать объекта класса Permissions можно следующим образом:

```
BankAccount.Permissions perm=
                   new BankAccount.Permissions();
```

У вложенных статических классов могут быть любые модификаторы доступа public, protected, private.

Следует заметить, что вложенный статический класс имеет доступ только к статическим атрибутам объемлющего класса:

```
class BankAccount{
 private long number;
 private static long balance;
 static class Permissions  {
  public boolean canDeposit, canwithdraw,
                                        canclose;
  public void func(){
  balance=10; //верно
  number=20; //error  };}
}
```

Не существует каких-либо ограничений, связанных с возможностью расширения вложенного статического класса, — класс может быть наследован любым другим классом, обладающим необходимыми правами доступа.

При этом, разумеется, производный класс не способен унаследовать те привилегии доступа к членам внешнего класса, которыми наделен вложенный класс.

Вложенные интерфейсы

Вложенные интерфейсы всегда "статичны", хотя соответствующий модификатор static может быть опущен.

Нестатические вложенные классы

Нестатические вложенные классы принято называть внутренними классами. Объект внутреннего класса всегда ассоциируется с соответствующим объектом внешнего класса. Рассмотрим пример:

```
public class BankAccount {
 private long number;
 private long balance;
 private Action lastAct;
   public class Action  {
     private String act;
     private long amount;
    Action(String act,   long amount)  {
            this.act = act;
            this.amount = amount;     }
        public String toString() {
                return number + ": " + act + " " + amount; }
  }
 public void deposit(long amount) {
            balance += amount;
            lastAct = new Action(“приход", amount);    }
  public void withdraw(long amount) {
            balance -= amount;
            lastAct = new Action("расход", amount);     } }
```

Объект внутреннего класса по умолчанию получает в свое распоряжение ссылку this на текущий внешний объект. В методе deposit, строку в которой создается объект Action, можно переписать в виде:

lastAct = this.new Action(“приход", amount);

Место this в подобном случае может занять, если это необходимо, ссылка на любой другой объект класса BankAccount.

Рассмотрим пример:

```
public void transfer(BankAccount other,  long amount){
   other.withdraw(amount) ;
   deposit(amount);
   lastAct = this.new Action("перевод",   amount);
   other.lastAct = other.new Action("перевод",  amount);
}
```

Можно создавать объекты внутреннего класса не только в объемлющих классах:

```
class BankAccount{
 private long number;
 private static long balance;
 class Permissions  {
   public boolean canDeposit, canwithdraw, canclose;
   public void func(){balance=10; };
 }
}
public class Main {
  public static void main(String[] args) {
BankAccount p1=new BankAccount();
BankAccount.Permissions p=p1.new Permissions();}}
```

Расширение внутренних классов

Внутренние классы допускают наследование. Единственное требование состоит в том, что объекты расширенного класса должны сохранять связь с объектами исходного внешнего класса или класса, производного от него.

Например:

```
class Outer { // внешний  класс
   class  Inner {….}  // внутренний  класс
 }
class ExtendedOuter extends Outer {
  // Расширенный внешний класс
  class ExtendedInner extends inner {….}
   // расширенный внутренний класс
  public inner ref = new ExtendedInner(); }
```

Поле ref инициализируется при создании объекта класса ExtendedOuter.

В ходе процесса построения экземпляра класса ExtendedInner вызывается его конструктор по умолчанию без параметров, который, в свою очередь, посредством ссылки super неявно обращается к конструктору по умолчанию класса Inner.

Конструктор Inner требует наличия объекта Outer, к которому следует "привязаться", — в этой роли выступает текущий объект класса ExtendedOuter.

Если внешним по отношению к расширенному внутреннему классу служит другой класс, не производный от Outer, либо если внутренний класс в результате расширения перестает быть внутренним, для обеспечения корректности вызова конструктора класса Inner с помощью ссылки super должна быть предоставлена дополнительная явная ссылка на объект Outer.

Например:

```
class Unrelated extends Outer.Inner {
 public Unrelated(Outer  ref)  {
  ref.super();  } }
```

При обращении во вложенных классах к полям базовых классов их следует снабжать ссылкой this или super.

Рассмотрим пример:

```
class C{
  int x=20  }
class A{
  int x=30;
  class B extends C{
    void increment(){
        x++; //x из класса C
        this.x++; //x из класса C
        super.x++; //x из класса C
        A.this.x++; // x из класса A
      };
    }
 }
```

Локальные внутренние классы

В Java разрешается объявлять вложенные классы внутри блоков кода, таких как тело метода, конструктора и т.д.

Единственный модификатор, который разрешено употреблять в объявлении локального класса – это final.

Код локального внутреннего класса обладает правом доступа ко всем переменным, принадлежащим контексту того же блока – локальным переменным, параметрам метода и т.д.

Единственное ограничение – локальная переменная или параметр метода становятся доступными, если они помечены как final.

Рассмотрим пример:

```
public static Iterator walkThrough(final  Object[]   objs) {
  class  iter implements  iterator {
     private int pos = 0;
     public boolean hasNext() {
              return(pos < objs.length);
      }
      public Object next() throws NoSuchElementException {
             if (pos >= objs.length)
                  throw new  NoSuchElementException();
             return objs[pos++];
      }
      public void remove() {
             throw new UnsupportedOperationException();
      }
  }
  return new Iter();
}
```

```
public class Main {
  public static void main(String[] args) {
    final int a=10;
    int c=20;
      class A{
       public void func(){
         int b=a; //верно, a имеет final
         b=c; //error
       };
}
```

Анонимные внутренние классы

Анонимный внутренний класс – это класс без имени.

Рассмотрим пример:

```
public static iterator walkThrough(final  Object[]  objs){
  return new Iterator(){
   //далее идет описание анонимного класса, реализующего
  // интерфейс Iterator
      private int pos = 0;
      public boolean hasNext()  {return (pos < objs.length); }
      public Object next() throws NoSuchElementException{
            if (pos >= objs.length)
                   throw new NoSuchElementException();
      return objs[pos++]; }
      public void  remove()   {
                throw new UnsupportedOperationException();
       }
   };
}
```

Вложенность в интерфейсах

Можно объявлять вложенные классы внутри интерфейсов. Рассмотрим пример:

```
interface A{
  class B{
    private int a=10;
   ………………
  }
 ……..
}
```

С помощью вложенных классов можно, в интерфейсе, создать поле, которое допускает изменение.

Рассмотрим пример:

```
interface A{
   class B{
           private int x=0;
           public int get(){return x;}
           public void setX(int n){x=n;}
   }
   B obj=new B();
 }
class C implements A{
  void func(){
     obj.setX(20);
    System.out.println(obj.get());
   }
}
public class Main {
  public static void main(String[] args) {
    C pc=new C();
    pc.func();
  }
} //На экране 20
```

В любом коде, реализующем интерфейс A возможен совместный доступ к данным посредством obj.