Advanced Programming Methods

Lecture 2 – Java Basics, Exceptions, Packages

Lecture Overview

1. Interfaces – completions
2. Downcasting
3. The operator instanceOf
4. The class Object
5. Packages
6. Access Modifiers
7. Exceptions
8. Introduction to Generics

Interfaces

- Prior to java 8, interfaces are designed to define a contract. They had only abstract methods and final constants.

- With Java 8, interfaces can have default and static methods (definitions).
- Default methods can be overridden. Whereas, static methods can’t be overridden.
- All the classes implementing interface should provide definition of all the methods in the interface.

Interfaces

- if the designers had to change the interface, then all the implementing classes are affected. They all have to provide a definition for the new method.
- Java 8's interface default and static methods provide a way to overcome this problem.

(an example from https://examples.javacodegeeks.com/interface-vs-abstract-class-java-example/)

```
public interface MobileInterface {

    /**
     * Java8 adds capability to have static method in interface.
     * Static method in interface Can't be overridden
     */
    public static void printWelcomeMessage() {
        System.out.println("***STATIC METHOD*** Welcome!!");
    }
    /*
     * Java8 adds capability of providing a default definition of method in an interface.
     * Default method can be overridden by the implementing class
     */
    default void makeACall(Long number) {
        System.out.println(String.format("***DEFAULT METHOD*** Calling ..... %d", number));
    }
    /*
     * Regular interface method, which every class needs to provide a definition
     */
    public void capturePhoto();
}
```

```
/**
 * BasicMobile class provides definition only for mandatory abstract method
 * Need to provide definition for capturePhoto()
 * Whereas, makeACall() takes the default implementation
 */

class BasicPhone implements MobileInterface {
    @Override
    public void capturePhoto() {
        System.out.println("***BASIC PHONE*** Cannot capture photo");
    }
}

/**
 * SmartPhone class overrides both default method and abstract method
 * Provides definition for both makeACall() and capturePhoto() methods
 */
class SmartPhone implements MobileInterface {
    @Override
    public void makeACall(Long number) {
        System.out.println(String.format("***SMART PHONE*** Can make voice and video call to
number .... %d", number));
    }
    @Override
    public void capturePhoto() {
        System.out.println("***SMART PHONE*** Can capture photo");
    }
}
```

```
/**
 * MobileInterfaceDemo is the driver class
 */
public class MobileInterfaceDemo {
    public static void main(String[] args) {
        MobileInterface basicPhone = new BasicPhone();
        MobileInterface smartPhone = new SmartPhone();

        // Calls static method of interface
        MobileInterface.printWelcomeMessage();
        System.out.println("********************** BASIC PHONE ************************");
        // Calls default implementation of interface
        basicPhone.makeACall(1234567890L);
        // Calls abstract method of interface
        basicPhone.capturePhoto();

        System.out.println("********************** SMART PHONE ************************");
        // Calls overridden implementation of makeACall()
        smartPhone.makeACall(1234567890L);
        // Calls abstract method of interface
        smartPhone.capturePhoto();
    }
}
```

Downcasting

```
Class Base {…}
Class SubBase extends Base{…}
Base objB=new Base();
SubBase objSB1 = new SubBase();
SubBase objSB2;
objB = objSB1; //correct
objSB2 = objB; //compiler rejects it
objSB2 = (SubBase) objB; //OK
//compiler does not verify it and trusts the
   programmer
```

Instanceof operator

- compares an object to a specified type.

- to test if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements a particular interface.

- null is not an instance of anything.

Instanceof operator

```
class Parent {}

class Child extends Parent implements MyInterface {}

interface MyInterface {}


Parent obj1 = new Parent();

Parent obj2 = new Child();


obj1 instanceof Parent: true

obj1 instanceof Child: false

obj1 instanceof MyInterface: false

obj2 instanceof Parent: true

obj2 instanceof Child: true

obj2 instanceof MyInterface: true
```

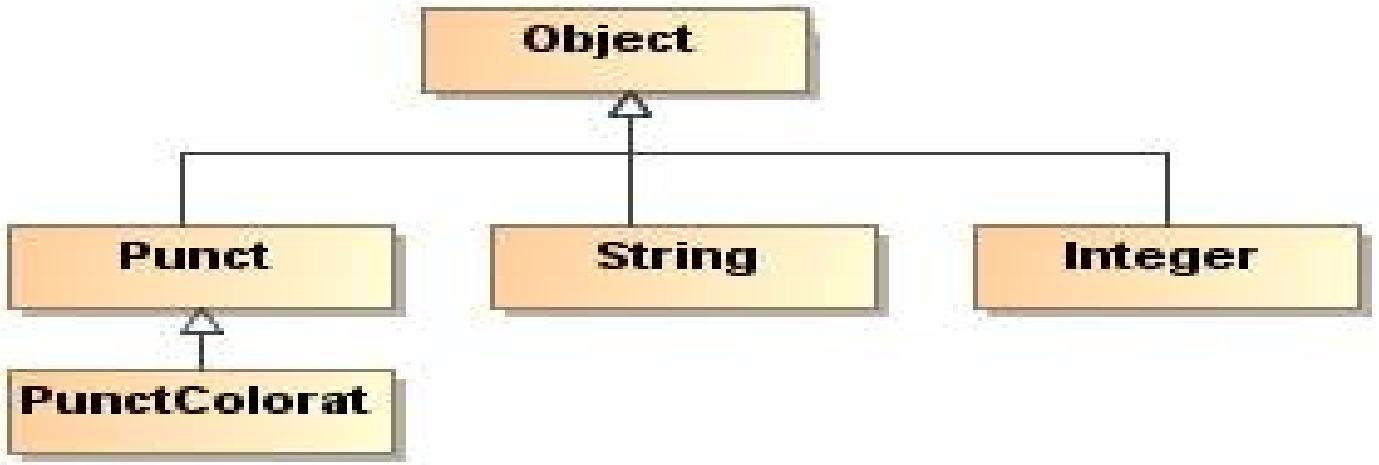
Real use of Instanceof operator

```
interface Printable{}
class A implements Printable{
public void a(){System.out.println("a method");}
}
class B implements Printable{
public void b(){System.out.println("b method");}
}
class Call{
void invoke(Printable p){
if(p instanceof A){
A a=(A)p;//Downcasting
a.a();
}
if(p instanceof B){
B b=(B)p;//Downcasting
b.b();
} } }
```

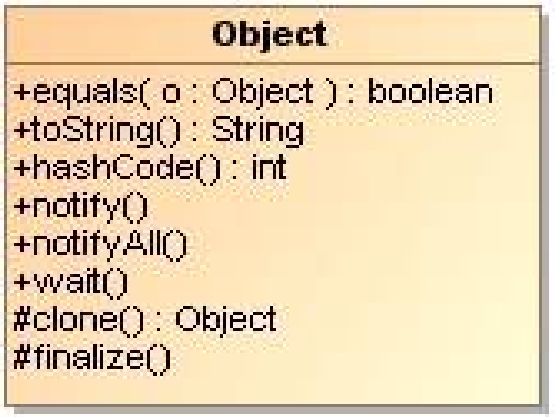
The class Object

- It is the top of the java classes hierarchy.
- By default Object is the parent of a class if other parent is not explicitly defined

```
public class Punct{
  //...
}
public class PunctColorat extends Punct{
  //...
}
```



Class Object - methods



- `toString`() is called when a String is expected
- `equals`() is used to check the equality of 2 objects. By default it compares the references of those 2 objects.

```
Punct p1=new Punct(2,3);
Punct p2=new Punct(2,3);
boolean egale=(p1==p2);   //false;
egale=p1.equals(p2);     //true, Punct must redefine equals
System.out.println(p1);   //toString is called
```

Overriding Class Object methods

```
public class Punct {
    private int x,y;
    public Punct(int x, int y) {
        this.x = x;
        this.y = y;
    }
    @Override
    public boolean equals(Object obj) {
        if (! (obj instanceof Punct))
            return false;
        Punct p=(Punct)obj;
        return (x==p.x)&& (y==p.y);
    }

    @Override
    public String toString() {
        return ""+x+' '+y;
    }
    //...
}
```

Packages

- Groups classes and interfaces
- Name space management

- ex. package `java.lang` contains the classes `System, Integer, String`, etc.
- A package is defined by the instruction `package:`

```
//structuri/Stiva.java
package structuri;
public interface Stiva{
   //...
}
```

Obs:

1. `package` must be the first instruction of the java file
2. The file is saved in the folder `structuri` (case-sensitive).

```
//structuri/liste/Lista.java
package structuri.liste;     //folder structuri/liste/Lista.java
public interface Lista{
   //...
}
```

Using the classes declared in the packages

```
// structuri/ArboreBinar.java
package structuri;
public class ArboreBinar{
  //...
}
```

- The classes are referred using the following syntax:

```
  [pac1.[pac2.[...]]]NumeClasa
//TestStructuri.java
public class TestStructuri{
  public static void main(String args[]){
    structuri.ArboreBinar ab=new structuri.ArboreBinar();
    //...
  }
}
```

Using the classes declared in the packages

- Instruction `import`:
  - one class:

```
import pac1.[pac2.[...]]NumeClasa;
```

  - All the package classes, but not the subpackages:

```
import pac1.[pac2.[...]]*;
```

- A file may contain multiple import instructions. They must be at the beginning before any class declaration.

```
//structuri/Heap.java
package structuri;
public class Heap{
  //...
}
//Test.java
//without import instructions
public class Test{
  public static void main(String args[]){
    structuri.ArboreBinar ab=new structuri.ArboreBinar();
    structuri.Heap hp=new structuri.Heap();
  }}
```

Using the classes declared in the packages

```
//Test.java
import structuri.ArboreBinar;
public class Test{
  public static void main(String args[]){
    ArboreBinar ab=new ArboreBinar();
    structuri.Heap hp=new structuri.Heap();
  }
}
//Test.java
import structuri.*;
import structuri.liste.*;
public class Test{
  public static void main(String args[]){
    ArboreBinar ab=new ArboreBinar();
    Heap hp=new Heap();
    Lista li=new Lista();
  }
}
```

Package+import

- The instruction `package` must be before any instruction `import`

```
//algoritmi/Backtracking.java
package algoritmi;
import structuri.*;

public class Backtracking{
  //...
}
```

- The package `java.lang` is implicitly imported by the compiler.

Anonymous package

```
//Persoana.java
public class Persoana{...}

//Complex.java
class Complex{...}

//Test.java
public class Test{
  public static void main(String args[]){
    Persoana p=new Persoana();
    Complex c=new Complex();
    //...
  }
}
```

If a file `.java` does not contain the instruction `package`, all the file classes are part of anonymous package.

Name Collision

```
// unu/A.java
package unu;
public class A{
  //...
}
```

```
// doi/A.java
package doi;
public class A{
  //...
}
```

```
//Test.java
import unu.*;
import doi.*;
public class Test{
  public static void main(String[] args){
    A a=new A();  //compilation error
    unu.A a1=new unu.A();
    doi.A a2=new doi.A();
  }
}
```

Access modifiers

- 4 modifiers for the class members:
  - `public:` access from everywhere
  - `protected:` access from the same package and from subclasses
  - `private:` access only from the same class
  - `:` access only from the same package
- Classes (excepting inner classes) and interfaces can be public or nothing.

Access modifiers

```
// structuri/Nod.java
package structuri;
class Nod{
   private Nod urm;
   public Nod getUrm(){...}
   void setUrm(Nod p){...}
  //...
}
```

```
// structuri/Coada.java
package structuri;
public class Coada{
   Nod cap;
   Coada(){ cap.urm=null;}
   Coada(int i){...}
  //...
}
```

```
//Test.java
import structuri.*;
class Test{
  public static void main(String args[]){
      Coada c=new Coada();
      Nod n=new Nod();     //class is not public
      Coada c2=new Coada(2);   //constructor is not public
 }
}
```

Access modifiers

```
package unu;
public class A{
  A(int c, int d){...}
  protected A(int c){...}
  public A(){...}
  protected void f(){...}
  void h(){...}
}
```

```
package unu;
class DA extends A{
   DA(int c){ super(c);}
}
```

```
package doi;
import unu.*;
class DDA extends A{
  DDA(int c){super(c);}
  DDA(int c, int d) {super(c,d);}
  protected void f(){
    super.h();
  }
}
```

Protected

- The fields and methods which are declared `protected` are visible inside the class, inside the derived classes and inside the same package.

```
public class Persoana{
  private String nume;
  private int varsta;
  public Persoana(String nume, int varsta){
    this.nume=nume;
    this.varsta=varsta;
  }
  //...
}
public class Angajat extends Persoana{
  private String departament;
  public Angajat(String nume, int varsta, String departament){
    this.nume=nume;
    this.varsta=varsta;
    this.departament=departament;
  }
  //...
}
```

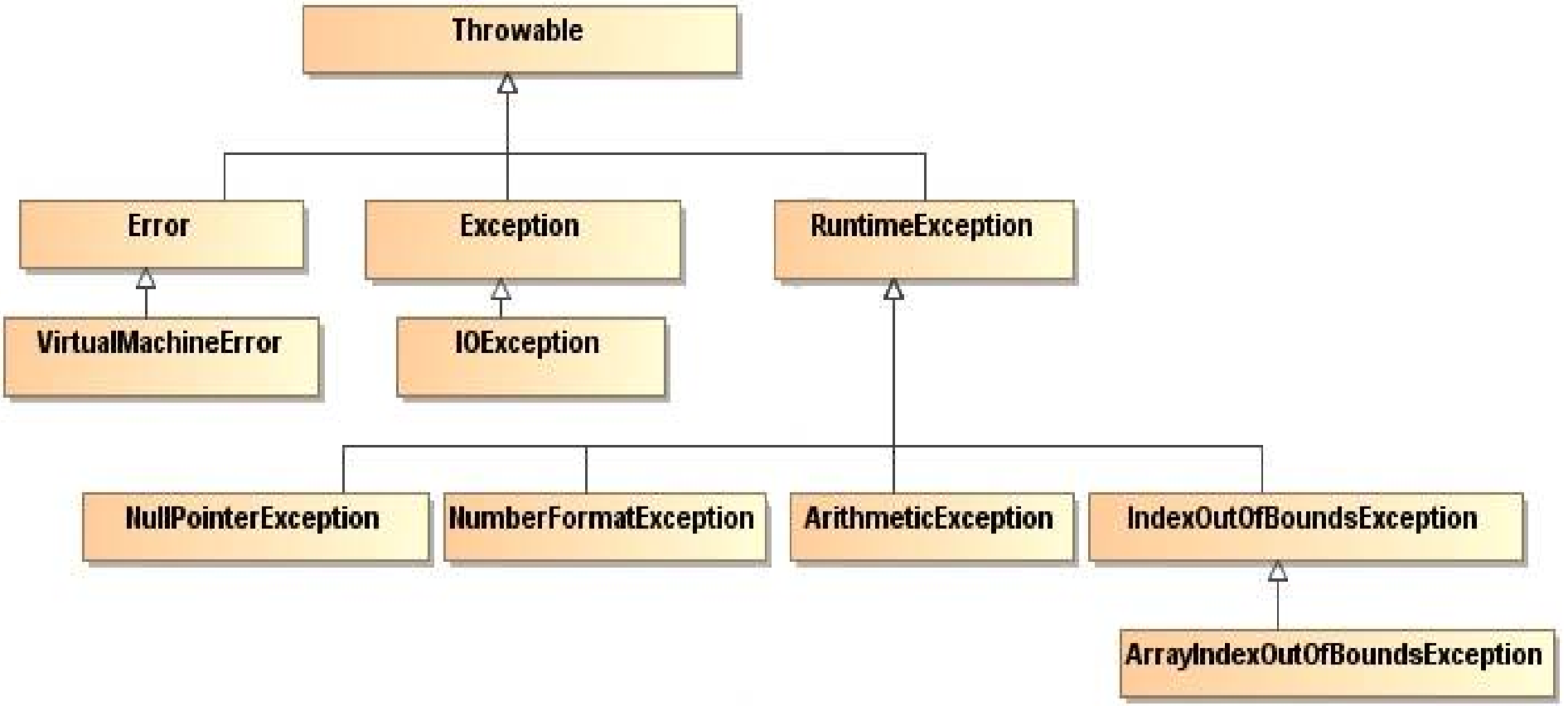
Protected

```
public class Persoana{
  protected String nume;
  protected int varsta;
  public Persoana(String nume, int varsta){
    this.nume=nume;
    this.varsta=varsta;
  }
  //...
}
public class Angajat extends Persoana{
  protected String departament;
  public Angajat(String nume, int varsta, String departament){
    this.nume=nume;
    this.varsta=varsta;
    this.departament=departament;
  }
  //...
}
```

JAVA EXCEPTIONS

Java Exceptions

Three types of exceptions:Errors(external to the application), Checked Exceptions(subject to try-catch), and Runtime Exceptions(correspond to some bugs)



Example 1

Program for ax+b=0, where a, b are integers.

```
class P1{
 public static void main(String args[]){
      int a=Integer.parseInt(args[0]);    //(1)
      int b=Integer.parseInt(args[1]);    //(2)
      if (b % a==0)                       //(3)
      System.out.println("Solutie "+(-b/a));          //(4)
      else
        System.out.println("Nu exista solutie intreaga"); //(5)
    }
}
java P1 1 1 //-1
java P1 0 3 //exception, divide by 0
            //Lines 4 or 5 are not longer executed
```

Example 1

Java VM creates the exception object corresponding to that abnormal situation and throws the exception object to those program instructions that generates the abnormal situation.

Thrown exception object can be caught or can be ignored (in our example the program P1 ignores the exception)

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
 at P1.main(P1.java:13)
```

Catching exceptions

Using try-catch statement:

```
try{
  //code that might generates abnormal situations
}catch(TipExceptie numeVariabila){
  //treatment of the abnormal situation
}
```

Execution Flow:

- If an abnormal situation occurs in the block try(), JVM creates an exception object and throws it to the block catch.
- If no abnormal situation occurs, try block normally executes.
- If the exception object is compatible with one of the exceptions of the catch blocks then that catch block executes

Example 2

```
class P2{
    public static void main(String args[]){
     try{
      int a=Integer.parseInt(args[0]);    //(1)
      int b=Integer.parseInt(args[1]);    //(2)
      if (b % a==0)                //(3)
        System.out.println("Solutie "+(-b/a));    //(4)
      else
        System.out.println("Nu exista solutie intreaga"); //(5)
     }catch(ArithmeticException e){
        System.out.println("Nu exista solutie");  //(6)
     }
    }
}
java P2 1 1 //Solutie -1
java P2 0 3 // Nu exista solutie
            // (1), (2), (3), (6) are executed
```

Multiple catch clauses

```
try{
  //code with possible errors
}catch(TipExceptie1 numeVariabila1){
  //instructions
}catch(TipExceptie2 numeVariabila2){
  //instructions
}...
catch(TipExceptien numeVariabilan){
  // instructions
}
```

Example 3

```
class P3{
    public static void main(String args[]){
     try{
      int a=Integer.parseInt(args[0]);    //(1)
      int b=Integer.parseInt(args[1]);    //(2)
      if (b % a==0)                //(3)
        System.out.println("Solutie "+(-b/a));     //(4)
      else
        System.out.println("Nu exista solutie intreaga"); //(5)
     }catch(ArithmeticException e){
        System.out.println("Nu exista solutie");  //(6)
     }catch(ArrayIndexOutOfBoundsException e){
        System.out.println("java P3 a b");      //(7)
     }    }
}
java P3 1 1 //Solution -1
java P3 0 3 // Nu exista solutie
            // (1), (2), (3), (6) are executed
java P3 1  //java P3 a b
```

Nested try statements

```
class P4{
  public static void main(String args[]){
    try{
      int a=Integer.parseInt(args[0]);    //(1)
      int b=Integer.parseInt(args[1]);    //(2)
      try{
         if (b % a==0)                 //(3)
          System.out.println("Solutie "+(-b/a));          //(4)
         else
          System.out.println("Nu exista solutie intreaga"); //(5)
      }catch(ArithmeticException e){
          System.out.println("Nu exista solutie");  //(6)
      }
      }catch(ArrayIndexOutOfBoundsException e){
        System.out.println("java P4 a b");       //(7)
       } }}
java P4 1 1 //Solutie -1
java P4 0 3 // Nu exista solutie
java P4 1  //java P4 a b
```

Finally clause

The finally clause is executed in any situation:

```
try{
  //...
}catch(TipExceptie1 numeVar1){
  //instructiuni
}[catch(TipExceptien numeVarn){
  // ...
}]
[finally{
  //instructiuni
}]
```

Finally Clause

```
A
try{
   B
}catch(TipExceptie nume){
    C
}finally{
  D
}
E
```

Block D executes:

- After A and B (before E) if no exception occurs in B. (A, B, D, E)
- After C, if an exception occurs in B and that exception is caught (A, a part of B, C, D, E).
- Before exit from the method:
  - » An exception occurs in B, but is not caught (A, a part of B, D).
  - » An exception occurs in B, it is caught but a return exists in C (A, a part of B, C, D).
  - » If a return exists in B (A, B, D).

Finally Clause

```
public void writeElem(int[] vec) {
    PrintWriter out = null;
    try {
        out = new PrintWriter(new FileWriter("fisier.txt"));
         for (int elem:vec)
             out.print(" "+elem);
    } catch (IOException e) {
         System.err.println("IOException: "+e);
    }finally{
         if (out != null)
             out.close();
    }
}
```

Defining exception classes

- By deriving from class `Exception:`

```
public class ExceptieNoua extends Exception{
  public ExceptieNoua(){}
  public ExceptieNoua(String mesaj){
    super(mesaj);
  }
}
```

Exceptions Specification

- Use keyword `throws` in method signatures:

```
public class ExceptieNoua extends Exception{}

public class A{
  public void f() throws ExceptieNoua{
    //...
  }
}
```

- Many exceptions can be specified (their order does not matter):

```
public class Exceptie1 extends Exception{}
public class B{
  public int g(int d) throws ExceptieNoua, Exceptie1{
    //...
 }
}
```

Throwing exceptions

- Statement `throw :`

```
public class B{
  public int  g(int d) throws ExceptieNoua,Exceptie1{
    if (d==3)
        return 10;
    if (d==7)
        throw new ExceptieNoua();
    if (d==5)
        throw new Exceptie1();
    return 0;
  }
  //...
}
```

- Statement throw throws away the exception object and the method execution is interrupted.
- All exceptions thrown inside a method must be specified in the method signature.

Calling a method having exceptions

- use try-catch to treat the exception:

```
public class C{
  public void h(A a){
    try{
      a.f();
    }catch(ExceptieNoua e){
      System.err.println(" Exceptie "+e);
    }
  }
}
```

- Throwing away an uncaught exception (uncaught exception must be specified in the signature):

```
public class C{
  public void t(B b) throws ExceptieNoua {
    try{
      int rez=b.g(8);
    }catch(Exceptie1 e){          //only Exceptie1 is caught
      System.err.println(" Exceptie "+e);
    }
  }
}
```

Exception specification

- The subclass constructor must specify all the base class constructor (explicitly or implicitly called) in its signature.
- The subclass constructor may add new exceptions to its signature.

```
public class A{
  public A() throws Exceptie1{
  }
  public A(int i){ }
  //...
}

public class B extends A{
  public B() throws Exceptie1{  }
  public B(int i){
    super(i);
  }
  public B(char c) throws Exceptie1, ExceptieNoua{
  }
  //...
}
```

Exceptions and method overriding

- An overriding method may declare a part of the exceptions of the overridden method.
- An overriding method may add only new exceptions which are inherited from the overridden method exceptions
- The same rules are applied for the interfaces.

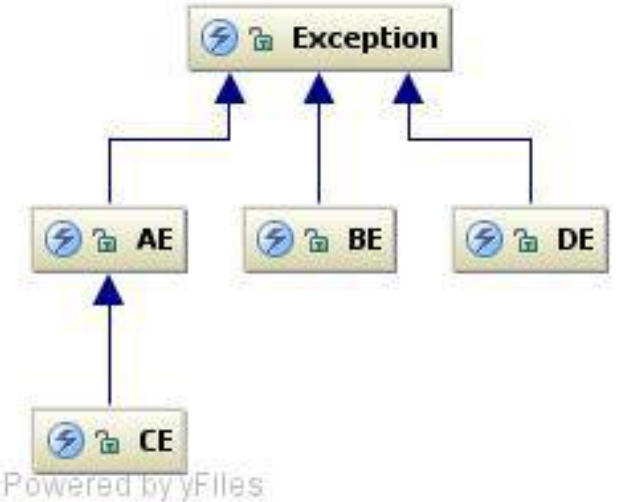
```
public class AA {
    public void f() throws Exceptie1, Exceptie2{ }
    public void g(){ }
    public void h() throws Exceptie1{  }
}

public class BB extends AA{
    public void f() throws Exceptie1{ }  //Exceptie2 is not declared
    public void g() throws Exceptie2{ }  //not allowed
    public void h() throws Exceptie3{ }
}
public class Exceptie3 extends Exceptie1{...}
```

Exceptions and method overriding

```
public class A {
    public void f() throws AE, BE {}
    public void g() throws AE{}
}

public class B extends A{
    public void g(){}
    public void f() throws AE, BE, CE{}
    public void f() throws AE, BE, DE{}
    public void g() throws DE{}
}
//?
```



Exceptions order in catch clauses

- The order of catch clauses is important since the JVM selects the first catch clause on which the try block thrown exception matches.
- An exception A matches an exception B if A and B have the same class or A is a subclass of B.

```
public class C {
  public void g(B b){
    try{
        b.f();
    } catch(Exception e){...

    } catch (CE ce) { ...
    } catch (AE ae) {...
    } catch (BE be) {...
    }
  }
}
```

```
public class C {
  public void g(B b){
    try{
        b.f();
    } catch (CE ce) { ...
    } catch (AE ae) {...
    } catch (BE be) {...
    } catch(Exception e){...
    }
  }
}
```

Re-throwing an exception

- A caught exception can be re-thrown

```
public class C {
  public int h(A a) throws Exceptie4, BE {
        try {
            a.f();
        } catch (BE be) {
            System.out.println("Exceptie rearuncata "+be);
            throw be;
        } catch (AE ae) {
            throw new Exceptie4("mesaj", ae);
        }
        return 0;
    }
}
public class Exceptie4 extends Exception {
    public Exceptie4() { }
    public Exceptie4(String message) {
        super(message);
    }
    public Exceptie4(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Introduction to Generics

A non generic class

```
public class SingleBox {
    private Object first;
    public void setFirst(Object object) { this.first = object; }
    public Object getFirst() { return first; }}
```

Class usage:

```
SingleBox ob= new SingleBox();
Integer i = new Integer(10);
ob.setFirst(i);//information about Integer is lost

//we need a downcast to recover the specific information, namely the Integer
Integer a = (Integer) ob.getFirst();
//the cast may be wrong, it will pass the compiler but a runtime error will occur
String s = (String) ob.getFirst();
```

First generic version

```
public class SingleBox<T> {
    // T stands for "Type"
    protected T first;
    public void setFirst(T t) { this.first = t; }
    public T getFirst() { return first; }}
```

Class usage:

```
SingleBox<Integer> ob= new SingleBox<Integer>();
Integer i = new Integer(10);
ob.setFirst(i);
Integer a = ob.getFirst();
```

String s = ob.getFirst(); //an error will be generated at compile time

Inheritance Example

```
public class PairBox<T1, T2> extends SingleBox<T1> {
    private T2 second;
    public PairBox(T1 f, T2 s) {
        this.first = f;  this.second = s;
    }

    public T2 getSecond()      { return second; }
    public void setSecond(T2 t) { this.second=t; }
}
```