Programming with Java

exceptions and interfaces

P. Bakowski

bako@ieee.org

The exceptions

The error handling mechanism is based exceptions. When a function is not defined for some values of its arguments it raises an exception using the **throw** keyword. For example, the factorial function is not defined for negative numbers, and in these cases it throws an exception:

```
class Factorial {
static int factorial(int n){
int res = 1;
if(n<0){
throw new NotDefined();
}
for(int i=1;i<=n;i++) {res = res*i; }
return res;
}
}
class NotDefined extends Error{}
```

How to define an exception ?

To define a new kind of exception, we create a new class by using a statement of the following form:

```
class NewException extends ExceptionAlreadyDefined{}
```

In this construction, **`NewException`** is the name of the exception class that you want to define as "extention" of **`ExceptionAlreadyDefined`** that is a class already defined. Knowing that **`Error`** is predefined in Java, the following statement sets the new exception class **`NotDefined`**:

```
class NotDefined extends Error{}
```

How to define an exception ?

Three categories of predefined exceptions in Java:

- Extension of the class **`Error`** - Critical Error

- Extention of **`Exception`** class - errors that must be handled by the program

- Extension of the class **`RuntimeException`** - errors possibly managed by the program.

Each new exception belogs to one of these three categories. The exception **`NotDefined`** would be declared by:

```
class NotDefined extends Exception{}
```

or by :

```
class NotDefined extends RuntimeException{}
```

because it is not a critical error.

How to throw an exception ?

When you want to throw an exception, use the `throw` keyword followed by the exception to be activated. The exception must be previously created with the construction of `new ExceptionName()`.

So to throw an exception of class `NotDefined` you write:

```
throw new NotDefined();
```

When an exception is raised, normal execution of the program stops and skips all instructions until the exception is caught or until it exits the program.

Throwing an exception

```
public class Stop {
  public static void main(String[] args){
  int x = Keyboard.readInt();
  System.out.println("Hi 1");
  if (x>0){
    throw new StopException();
    }
  System.out.println("Hi 1");
  System.out.println("Hi 2");
  System.out.println("Hi 3");
  }
}
public class StopException extends
RuntimeException{}
```

```
Java/Test>java Stop
Hi 1
Exception in thread "main" Stop
at Stop.main(Arret.java:7)
```

Catching an exception

The catching of an exception in Java is done using the construction:

```
try {
... // block 1
} catch ( AnException e ) {
... // 2
}
.. // 3
```

If the class of the thrown exception in block 1 is **`AnException`** then code 2 is executed, because the exception is caught by the **`catch`**.

Catching an exception

```
public class Stop2 {
  public static void P(){
  int x = Keyboard.readInt();
  if(x>0){ throw new StopException(); }
  }
  public static void main(String[] args){
  System.out.println("Hi 1");
  try {
     P();
     System.out.println("Hi 2");
     } catch(StopException e){System.out.println("Hi 3");}
  System.out.println("Hi 4");
  }
}
class StopException extends RuntimeException{}
```

if x>0	Hi 1 Hi 3 Hi 4
if x<0	Hi 1 Hi 2 Hi 4

Catching several exceptions

```
public class Stop3 {
  public static void P(){
  int x = Keyboard.readInt();
  if(x>0){ throw new StopException(); }
  else if (x==0) { throw new StopException0() ; }
  }
  public static void main(String[] args){
  try { P();System.out.println("Hi 2");}
  catch(StopException e){System.out.println("Hi 3");}
  catch(StopException0 e){System.out.println("Hi 4");}
  }
}
class StopException extends RuntimeException{}
class StopException0 extends RuntimeException{}
```

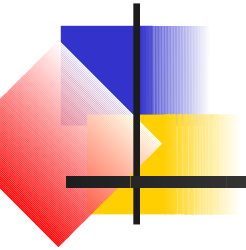
Si x>0	Hi 3	Si x==0	Hi 4	Si x<0	Hi 2

Declaration `throws`

When a method throws an exception defined by extending the **`Exception`** class, it is necessary to specify in the declaration of the method that it can potentially throw an exception in this class.
This statement takes the form **`throws`** `Exception1, Exception2`, and is placed between the arguments of the method and the opening brace marking the beginning of the method body.
`throws` is not required for the exceptions of **`Error`** class or for those of **`RuntimeException`** class.

```
static int factorial(int n) throws NotDefined{
int res = 1;
if(n<0){ throw new NotDefined();}
for(int i=1;i<=n;i++) { res = res*i;}
return res;
}

class NotDefined extends Exception{}
```



Some predefined exceptions

– **`NullPointerException`** : when using `length` or access method to a cell in the **`null`** array (not yet created by an **`new`**).

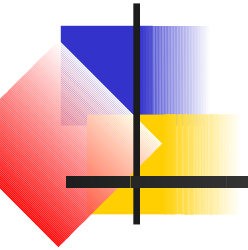
- **`ArrayIndexOutOfBoundsException`** : access to a cell not existing in the array.

- **`StringIndexOutOfBoundsException`** : access to a character of a String of smaller size then `i`.

- **`NegativeArraySizeException`** : attempt to create an array of negative size .

- **`NumberFormatException`** : error during the conversion ef a String into a number.

The class `Keyboard` also provides **`TerminalException`** to signal the errors.



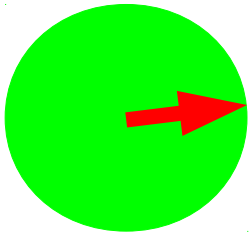
Interface

Interfaces are constructions of Java that allows to abstract behaviors of objects so that the objects belonging to different classes can be understood on the basis of similar components.

In this lecture, we will develop an example of geometric representations in 2D.
To start we will use : circles, rectangles and triangles.
Each type has specific form.
The data required to represent the forms are different. But all these forms have a surface.The formula to calculate the surface is different in each class, but the result is similar.
We can calculate the surface to know whether a circle is larger than a rectangle.

The forms without Interface

```
class Point {
double x,y;
  Point(double xi,double yi){x=xi; y=yi;}
  static double distance(Point p1,Point p2){
  return Math.sqrt((p1.x-p2.x)*(p1.x-p2.x)+
                   (p1.y-p2.y)*(p1.y-p2.y));}
}
```




```
class Circle {
Point center;
double radius;
  Circle (Point ctr, double r){center=ctr;radius=r;}
  double surface() {
  return Math.PI*radius*radius; }
}
```

The forms without Interface

```
class Rectangle {
Point bottomLeft;
double dimHor, dimVer;
  Rectangle(Point bg, double dh, double dv)
  { bottomLeft=bg; dimHor=dh; dimVer=dv; }
  double surface() {
  return dimHor*dimVer; }
}
```

dimVer

dimHor

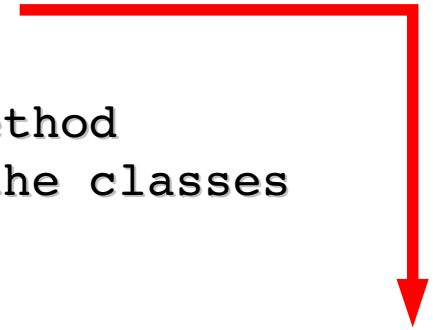
```
class Triangle {
Point p1,p2,p3;
double dimHor, dimVer;
  Triangle(Point p1i, Point p2i, Point p3i)
  { p1=p1i; p2=p2i; p3=p3i; }
  double surface() {
  double a = Point.distance(p1,p2);
  double b = Point.distance(p1,p3);
  double c = Point.distance(p2,p3);
  double dp = (a+b+c)/2;
  return Math.sqrt(dp*(dp-a)*(dp-b)*(dp-c));}
}
```

p1

p2 p3

The forms with Interface

```
interface WithSurface {
double surface();
// there is no body of method
// to be implementer in the classes
}
```



```
class CircleIntf implements WithSurface {
Point center;
double radius;
  CircleIntf(Point ctr, double r)
          {center=ctr;radius=r;}
  public double surface() {  //  public method !
  return Math.PI*radius*radius;
}
}
```

The forms with Interface

```
class RectangleIntf implements WithSurface {
Point bottomLeft;
double dimHor, dimVer;
  RectangleIntf(Point bg, double dh, double dv)
  { bottomLeft=bg; dimHor=dh; dimVer=dv; }
  public double surface() {
  return dimHor*dimVer; }
}
```

```
class TriangleIntf implements WithSurface {
Point p1,p2,p3;
double dimHor, dimVer;
  TriangleIntf(Point p1i, Point p2i, Point p3i)
  { p1=p1i; p2=p2i; p3=p3i; }
  public double surface() {
  double a = Point.distance(p1,p2);
  double b = Point.distance(p1,p3);
  double c = Point.distance(p2,p3);
  double dp = (a+b+c)/2;
  return Math.sqrt(dp*(dp-a)*(dp-b)*(dp-c));}
}
```

The forms with Interface

```
public class PlanFormsIntf {
public static void main(String[] args){
Point p1=new Point(1,3); Point p2=new Point(1,5);
Point p3=new Point(2,4);
WithSurface[] tab = new WithSurface[3];
tab[0] = new TriangleIntf(p1,p2,p3);
tab[1] = new RectangleIntf(p1,2.7,5.0);
tab[2] = new CircleIntf(p1,2.5);
for(int i=0;i<3;i++){
System.out.println("surface of tab["+i+"]:" +
tab[i].surface());
}
if((tab[0].surface()>tab[1].surface())&&
(tab[0].surface()>tab[2].surface())) {
System.out.println("tab[1] is larger");}
else {
System.out.println("tab[2] is the largest");}
}
}
```

Interface – some rules

- The interface contains only declaration of one or more methods, that is to say the type of parameters and results, and possibly the name of the exceptions thrown by the method.

- Classes that implement the interface must explicitly declare the name of the interface. For example : `class TriangleIntf implements WithSurface`

- The methods declared in the interface must be defined in the implementing classes with the same name, the same type for parameters and results. These methods must be defined with the keyword `public`. This keyword does not appear in the method declaration in the interface.

- A variable declared with the type the name of an interface can contain an instance of any class that implements the interface.

- A variable of this type, you can call all the methods in the interface, but no other methods.

The forms with two Interfaces

```
interface WithTranslation {// interface at the level of Point
void translation(double offsetHor, double offsetVer);
}
class Point implements WithTranslation {
  double x,y;
  Point(double xi,double yi){x=xi; y=yi;}
  static double distance(Point p1,Point p2){
  return Math.sqrt((p1.x-p2.x)*(p1.x-p2.x)+
                   (p1.y-p2.y)*(p1.y-p2.y));}
  public void translation(double offsetHor, double offsetVer)
  {x=x+offsetHor; y=offsetVer;}
}
```

```
interface WithSurface{ // interface at the level of forms
double surface() ;
}
```

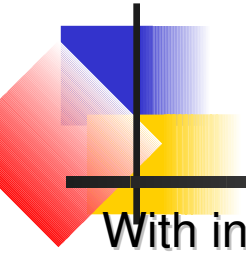
The forms with two Interfaces

```
interface WithSurface{ // interface at the level of forms
double surface() ;
}

class Circle implements WithSurface, WithTranslation {
Point center;
double radius;
  Circle (Point ctr, double r)
          {center=ctr;radius=r;}
  public double surface() {  // public method!
  return Math.PI*radius*radius;
  }
public void translation(double offsetHor, double offsetVer)
  {center.translation(offsetHor,offsetVer);}
}
```

The forms with two Interfaces

```
public class Plan3 {
public static void main(String[] args){
Point p1=new Point(1,3);Point p2=new Point(1,5);Point p3=new Point(2,4);
WithSurface[] tab = new WithSurface[3];
tab[0] = new TriangleIntf(p1,p2,p3);
tab[1] = new RectangleIntf(p1,2.7,5.0);
tab[2] = new CircleIntf(p1,2.5);
for(int i=0;i<3;i++){
System.out.println("surface of tab["+i+"]:" + tab[i].surface());}
WithTranslation[] tab2= new WithTranslation[4];
tab2[0] = new TriangleIntf(p1,p2,p3); /
tab2[1] = new RectangleIntf(p1,2.7,5.0);
tab2[2] = new CircleIntf(p1,2.5);
tab2[3] = new Point(1,5);              // implements translation()
for(int i=0;i<4;i++){ tab2[i].translation(2.0,0); }
}
}
```



History of types

With interfaces, an object has several Java types.

For example, an object instance of the `Circle` class will be placed in a variable of one of three types : `Circle`, `WithSurface`, and `WithTranslation`.

In a variable `Circle`, it will have two methods `surface()` and `translation()`. With the type `WithSurface`, it will have only the method `surface()` and type `WithTranslation`, it will only have the `translation()` method.

It is possible to assign an object of type `Circle` in a variable of type `WithSurface` because the `Circle` class implements the interface `WithSurface`.

In the other direction, the assignment is not possible without explicit type conversion, as objects of type `Circle` have features that not all objects of type `WithSurface`. For example, they both center and radius variables that are not necessarily objects of type `WithSurface`.

History of types

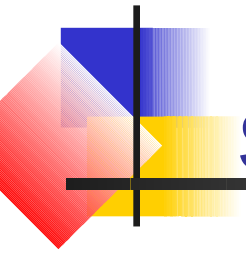
```
public class Conv {
  public static void main(String[] args){
  Point p=new Point(1,3);
  Circle c1 = new Cercle(p,2.3);
  WithSurface c2 = new Circle(p,5);
  WithTranslation c3 = new Cercle(p,4.2);
  c2 = c1 ; // OK
  c1 = c2 ; // compilation error
  c2 = c3 ; // compilation error
  }
}
```

« Abstract » programming

We want to compare the surface of two figures. With the type **WithSurface**, we can write a single method that accepts a parameter as well as for a `Circle` a `Rectangle` or a `Triangle`.

```
interface WithSurface { // interface au niveau des formes
double surface();
int compareSurface(WithSurface as); }

 class Circle implements WithSurface {
 Point center; double radius;
   Circle (Point ctr, double r) {center=ctr;radius=r;}
   public double surface() {
   return Math.PI*radius*radius;
   }
   public int compareSurface(WithSurface as){
   int res=0 ;
   if(this.surface()<as.surface()) res =-1 ;
   else res =1;  return res ;
   }
 // the same mathod implemented in Rectangle and Triangle
 }
```



Summary

Raise an exception(`throw`)

Define an exception : `Error`, `Exception`, `RuntimeException`

Catch an exception : `try {..} catch() {}`

Predefined Exceptions

Interfaces : `interface`, `implements`,

« Abstract » programming