Programmation Java

exceptions et interfaces

P. Bakowski

bako@ieee.org

Les exceptions

Le mécanisme de gestion d'erreur le plus répandu est celui des exceptions. Lorsqu'une fonction n'est pas définie pour certaines valeur de ses arguments on lève une exception en utilisant le mot clef **`throw`**. Par exemple, la fonction `factorielle` n'est pas définie pour les nombres négatifs, et pour ces cas, on lève une exception :

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

Définir des exceptions

Afin de définir une nouvelle sorte d'exception, on crée une nouvelle classe en utilisant une déclaration de la forme suivante :

```
class NouvelleException extends ExceptionDejaDefinie{}
```

Dans cette construction, `NouvelleException` est le nom de la classe d'exception que l'on désire définir en "étendant" `ExceptionDejaDefinie` qui est une classe d'exception déjà définie. Sachant que `Error` est prédéfinie en Java, la déclaration suivante définit la nouvelle classe d'exception `PasDefini` :

```
class PasDefini extends Error{}
```

Définir des exceptions

Trois catégories d'exceptions préd éfinies en Java:

- extension de la classe `Error` - erreurs critiques

- extension de la classe `Exception` - erreurs qui doivent etre gérées par le programme

- extension de la classe `RuntimeException` - erreurs eventuellement gérées par le programme.

Chaque nouvelle exception entre ainsi dans l'une de ces trois catégories.
L'exception PasDefini aurait due etre déclarée par :

```
class PasDefini extends Exception {}
```

ou bien par :

```
class PasDefini extends RuntimeException {}
```

car elle ne constitue pas une erreur critique.

Lever une exception

Lorsque l'on veut lever une exception, on utilise le mot clé **`throw`** suivi de l'exception a lever, qu'il faut avoir créee auparavant avec la construction **`new`** `NomException().`

Ainsi pour lancer une exception de la classe `PasDefini` s'écrit :

```
throw new PasDefini();
```

Lorsqu'une exception est levée, l'exécution normale du programme s'arrête et on saute toutes les instructions jusqu'à ce que l'exception soit rattrapée ou jusqu'à ce que l'on sorte du programme.

Lever une exception

```
public class Arret {
  public static void main(String[] args){
  int x = Terminal.lireInt();
  Terminal.ecrireStringln("Coucou 1");
  if (x>0){
    throw new Stop();
    }
  Terminal.ecrireStringln("Coucou 1");
  Terminal.ecrireStringln("Coucou 2");
  Terminal.ecrireStringln("Coucou 3");
  }
}
class Stop extends RuntimeException{}
```

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

Rattraper une exception

Le rattrapage d'une exception en Java se fait en utilisant la construction :

```
try {
... // bloc 1
} catch ( UneException e ) {
... // 2
}
.. // 3
```

Si la classe de l'exception levée dans le bloc 1 est `UneException` alors le code 2 est exécuté , car l'exception est récupérée par le `catch`.

Rattraper une exception

```
public class Arret2 {
  public static void P(){
  int x = Terminal.lireInt();
  if(x>0){ throw new Stop(); }
  }
  public static void main(String[] args){
  Terminal.ecrireStringln("Coucou 1");
  try {
     P();
     Terminal.ecrireStringln("Coucou 2");
     } catch(Stop e){Terminal.ecrireStringln("Coucou 3");}
  Terminal.ecrireStringln("Coucou 4");
  }
}
class Stop extends RuntimeException{}
```

Si x>0	Coucou 1 Coucou 3 Coucou 4	Si x<0	Coucou 1 Coucou 2 Coucou 4

Rattraper plusieurs exceptions

```
public class Arret2 {
  public static void P(){
  int x = Terminal.lireInt();
  if(x>0){ throw new Stop(); }
  else if (x==0) { throw new Stop0() ; }
  }
  public static void main(String[] args){
  try { P();Terminal.ecrireStringln("Coucou 2");}
  catch(Stop e){Terminal.ecrireStringln("Coucou 3");}
  catch(Stop0 e){Terminal.ecrireStringln("Coucou 4");}
  }
}
class Stop extends RuntimeException{}
class Stop0 extends RuntimeException{}
```

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

Déclaration `throws`

Lorsqu'une méthode lève une exception définie par extension de la classe `Exception` il est nécessaire de préciser au niveau de la déclaration de la méthode qu'elle peut potentiellement lever une exception de cette classe.
Cette déclaration prend la forme **`throws`** `Exception1, Exception2,` et se place entre les arguments de la méthode et l'accolade ouvrant marquant le début du corps de la méthode.
`throws` n'est pas obligatoire pour les exceptions de la catégorie `Error` ni pour celles de la catégorie `RuntimeException`.

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

class PasDefini extends Exception{}
```

Déclaration `throws`

Lorsqu'une méthode lève une exception définie par extension de la classe `Exception` il est nécessaire de préciser au niveau de la déclaration de la méthode qu'elle peut potentiellement lever une exception de cette classe.
Cette déclaration prend la forme **`throws`** `Exception1, Exception2,` et se place entre les arguments de la méthode et l'accolade ouvrant marquant le début du corps de la méthode.
`throws` n'est pas obligatoire pour les exceptions de la catégorie `Error` ni pour celles de la catégorie `RuntimeException`.

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

class PasDefini extends Exception{}
```

Quelques exceptions prédéfinies

– `NullPointerException` : utilisation de `length` ou accès à une case d'un tableau valant **`null`** (c'est a dire non encore crée par un **`new`**).
- `ArrayIndexOutOfBoundsException` : accès à une case inexistante dans un tableau.
- `StringIndexOutOfBoundsException` : accès au i-éme caractère d'une chaîne de caractères de taille inférieure à i.
- `NegativeArraySizeException` : création d'un tableau de taille négative.
- `NumberFormatException` : erreur lors de la conversion d'une chaîne de caractères en nombre.

La classe `Terminal` utilise egalement l'exception `TerminalException` pour signaler des erreurs.

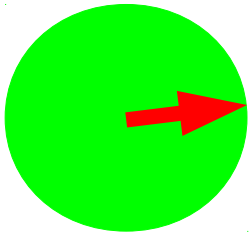
Interface

Les interfaces sont une construction du langage JAVA qui permettent d'abstraire certains comportements des objets de telle sorte que des objets appartenant à des classes différentes puissent être appréhendés sur la base de ce qu'ils ont de semblable.

Dans ce cours, nous allons développer un exemple de réprésentation des formes géométriques dans le plan.
Pour commencer, des cercles, rectangles et triangles.
Chaque type de forme a des spécificités.
Les données nécessaires pour les représenter sont différentes. Mais ces différentes formes ont en commun d'avoir une surface.
La formule de calcul est différente dans chaque classe, mais le résultat est comparable : on peut utiliser la surface pour savoir si un cercle est plus grand qu'un rectangle.

Les formes sans 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 Cercle {
Point centre;
double rayon;
  Cercle (Point ctr, double r){centre=ctr;rayon=r;}
  double surface() {
  return Math.PI*rayon*rayon; }
}
```

Les formes sans Interface

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


```
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));}
}
```

Les formes avec Interface

```
interface AvecSurface{
double surface();
// Pas de corps de methode
// a implementer dans les classes communes
}

        class Cercle implements AvecSurface {
        Point centre;
        double rayon;
          Cercle (Point ctr, double r)
                  {centre=ctr;rayon=r;}
          public double surface() {  // methode public !
          return Math.PI*rayon*rayon;
        }
        }
```

Les formes avec Interface

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

```
class Triangle implements AvecSurface {
Point p1,p2,p3;
double dimHor, dimVer;
  Triangle(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));}
}
```

Les formes avec Interface

```
public class Plan2 {
public static void main(String[] args){
Point p1=new Point(1,3);Point p2=new Point(1,5);Point p3=new Point(2,4);
Point p3=new Point(2,4);
Triangle t = new Triangle(p1,p2,p3);
Cercle c = new Cercle(p1,2.5);
Rectangle r = new Rectangle(p1,2.7,5.0);
AvecSurface as = t ;
AvecSurface[] tab = new AvecSurface[3];
tab[0] = new Triangle(p1,p2,p3);  // implementation par Triangle
tab[1] = new Rectangle(p1,2.7,5.0); // implementation par Rectangle
tab[2] = new Cercle(p1,2.5); // implementation par Cercle
for(int i=0;i<3;i++){
Terminal.ecrireStringln("surface de tab["+i+"]:" + tab[i].surface());
}
if((tab[0].surface()>tab[1].surface())&&
(tab[0].surface()>tab[2].surface())) {
Terminal.ecrireStringln("tab[1] est le plus grand");}
else {
Terminal.ecrireStringln("tab[2] est le plus grand");}
}
}
```

Interface – quelques règles

- l'interface contient seulement la déclaration d'une ou plusieurs méthodes, c'est a dire le type des paramètres et du résultat, le nom et eventuellement des exceptions levées par la méthode.

- les classes qui implémentent l'interface doivent le déclarer explicitement après le nom de la classe. Par exemple `class Triangle implements AvecSurface`.

- les méthodes déclarées dans l'interface doivent être définies dans les classes qui l'implémentent avec le même nom, le même type pour les paramètres et le résultat. Ces méthodes doivent etre définies avec le mot-clé `public`. Ce mot-clé ne doit pas apparaître dans la déclaration de la méthode dans l'interface.

- une variable déclarée avec pour type le nom d'une interface peut contenir une instance de n'importe quelle classe qui implémente l'interface.

- sur une variable de ce type, on peut appeler toutes les méthodes de l'interface, mais rien que ces méthodes.

Les formes avec deux Interfaces

```
interface AvecTranslation { // interface au niveau de Point
void translation(double deplHor, double deplVer);
}
class Point implements AvecTranslation {
  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 deplHor, double deplVer)
  {x=x+deplHor; y=y+deplHor;}
}


interface AvecSurface{ // interface au niveau des formes
double surface() ;
}
```

Les formes avec deux Interfaces

```
interface AvecSurface{ // interface au niveau des formes
double surface() ;
}

class Cercle implements AvecSurface, AvecTranslation {
Point centre;
double rayon;
  Cercle (Point ctr, double r)
         {centre=ctr;rayon=r;}
  public double surface() {  // methode public !
  return Math.PI*rayon*rayon;
  }
public void translation(double deplHor, double deplVer)
  {centre.translation(deplHor,deplVer);}
}
```

Les formes avec deux 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);
Point p3=new Point(2,4);
Triangle t = new Triangle(p1,p2,p3);
Cercle c = new Cercle(p1,2.5);
Rectangle r = new Rectangle(p1,2.7,5.0);
AvecSurface as = t ;
AvecSurface[] tab = new AvecSurface[3];
tab[0] = new Triangle(p1,p2,p3);     // Triangle implemente surface()
tab[1] = new Rectangle(p1,2.7,5.0); // Rectangle implemente surface()
tab[2] = new Cercle(p1,2.5);         // Cercle implemente surface()
for(int i=0;i<3;i++){
Terminal.ecrireStringln("surface de tab["+i+"]:" + tab[i].surface());}
AvecTranslation[] tab2= new AvecTranslation[4];
tab2[0] = new Triangle(p1,p2,p3); // Triangle implemente translation()
tab2[1] = new Rectangle(p1,2.7,5.0); // implemente translation()
tab2[2] = new Cercle(p1,2.5);         // implemente translation()
tab2[3] = new Point(1,5);             // implemente translation()
for(int i=0;i<4;i++){ tab2[i].translation(2.0,0); }
}
}
```

Histoire de types

Avec les interfaces, un objet possède plusieurs types Java.

Par exemple, un objet instance de la classe `Cercle` va pouvoir être placé dans une variable d'un des trois types `Cercle`, `AvecSurface` et `AvecTranslation`.

Dans une variable de type `Cercle`, il aura les deux méthodes `surface()` et `translation()`. Avec le type `AvecSurface`, il n'aura que la méthode `surface()` et avec le type `AvecTranslation`, il n'aura que la méthode `translation()`.

Il est possible de faire une affectation d'un objet de type `Cercle` dans une variable de type `AvecSurface` parce que la classe `Cercle` implémente l'interface `AvecSurface`.

Dans l'autre sens, l'affectation n'est pas possible sans conversion de type explicite, car les objets de type `Cercle` ont des caractéristiques que n'ont pas tous les objets du type `AvecSurface`. Par exemple, ils ont les deux variables `centre` et `rayon` que n'ont pas nécessairement les objets du type `AvecSurface`.

Histoire de types

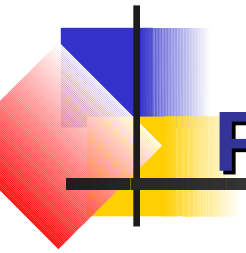
```
public class Conv {
  public static void main(String[] args){
  Point p=new Point(1,3);
  Cercle c1 = new Cercle(p,2.3);
  AvecSurface c2 = new Cercle(p,5);
  AvecTranslation c3 = new Cercle(p,4.2);
  c2 = c1 ; // OK
  c1 = c2 ; // erreur a la compilation
  c2 = c3 ; // erreur a la compilation
  }
}
```

Programmation « abstraite »

Nous voulons comparer la surface de deux Figures. Grâce au type `AvecSurface`, on peut écrire une seule méthode qui accepte en paramètre aussi bien un `Cercle` qu'un `Rectangle` ou un `Triangle`.

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

 class Cercle implements AvecSurface {
 Point centre; double rayon;
   Cercle (Point ctr, double r) {centre=ctr;rayon=r;}
   public double surface() {
   return Math.PI*rayon*rayon;
   }
   public int compareSurface(AvecSurface as){
   int res=0 ;
   if(this.surface()<as.surface()) res =-1 ;
   else res =1;  return res ;
   }
 // la même méthode implementée dans Rectangle et Triangle
 }
```



Résumé

Lever une exception (`throw`)

Définir une exception : `Error, Exception, RuntimeException`

Rattraper une exception : `try {..} catch() {}`

Exceptions prédéfinies

Interfaces : `interface`, `implements`,

Programmation « abstraite »