Chapitre 6

Exceptions

JAVA possède un mécanisme de gestion des erreurs, ce qui permet de renforcer la robustesse du code (i.e. son abilité à continuer de fonctionner malgré certains problèmes). Vous avez déjà dû voir le résultat de l'exécution d'une exception quand vous avez lu des messages d'erreur.

```
1 int[] tab = new int[5] ;
2 tab[5]=0 ;
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException : 5
at Personnage.main(Personnage.java :20)
```

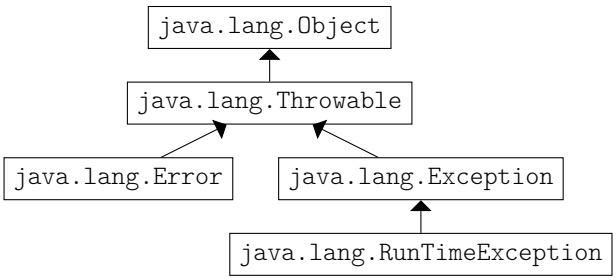
Dans l'exemple ci-dessus, le tableau est de taille 5, mais on essaie d'affecter la sixième entrée du tableau. JAVA indique la présence d'une exception. En fait, lorsqu'une erreur intervient, JAVA va instancier une classe qui correspond à l'erreur (on appelle cela lever une exception), puis va chercher à traiter l'erreur. Dans cet exemple, il s'agit d'une instance de la classe `java.lang.ArrayIndexOutOfBoundsException`. Le traitement de l'erreur est ici d'afficher le nombre 5, qui correspond à l'index du tableau que l'on cherche à accéder, mais qui n'existe pas. Ensuite, JAVA indique la méthode dans laquelle l'erreur s'est produite (ici dans la méthode `main` de la classe `Personnage`. Dans l'exemple suivant, on fait une division par 0 et on lève une exception de la classe `java.lang.ArithmeticException`, le traitement de l'exception fait simplement imprimer le message `/ by zero`.

```
1 int d=10,t1=5,t2=5 ;
2 System.out.println("vitesse :" + d / (t2-t1)) ;
```

```
Exception in thread "main" java.lang.ArithmeticException : / by zero
at Personnage.main(Personnage.java :21)
```

Dans ces exemples, l'application s'est terminée. Le but de la gestion des erreurs est de *capturer* l'exception, la traiter et ainsi parfois éviter l'arrêt du code. Certaines méthodes de JAVA sont susceptibles de produire des erreurs, et JAVA exige la prise en compte de ces erreurs potentielles (la compilation échoue si on ne le fait pas). Pour se faire, on peut faire appel à deux mécanismes : soit traiter l'exception en utilisant ce que l'on appelle un bloc `try ... catch` qui indique que faire en cas d'erreurs soit déléguer la gestion de l'erreur à la méthode appelante, dans ce cas on utilise le mot-clé `throws`. Nous allons voir cela un peu plus en détail.

La classe `Throwable` décrit l'ensemble des exceptions : à toute erreur correspond une instance d'une classe dérivée de `Throwable`. On peut avoir différents niveaux de problèmes :



La classe `Error` représente une erreur grave intervenue dans la machine virtuelle (par exemple `OutOfMemory`). Elle correspond à des erreurs matérielles ou à des erreurs de compilation. L'application JAVA s'arrête dès qu'une telle erreur est detectée, autrement dit, c'est la machine virtuelle de JAVA qui gère de telles erreurs et le programmeur ne peut pas les contrôler.

La classe `RunTimeException` regroupe aussi des exceptions qui ne peuvent être contrôlées par le programmeur.

Les autres exceptions qui dérivent de la classe `Exception` représentent des erreurs moins graves et le développeur a la possibilité de gérer de telles erreurs et ainsi éviter que l'application ne se termine, et c'est ce qui va nous occuper dans cette section.

Le traitement d'une erreur se décompose en trois étapes.

— La détection d'une erreur provoque l'instanciation d'un objet qui hérite de la classe `Exception` – c'est ce que l'on nomme lever une exception.
— la propagation (cette notion deviendra plus claire dans la section 2) : l'erreur peut être traitée « localement » ou peut être propagée récursivement à une méthode appelante grâce au mot-clé **`throw`**, i.e. l'erreur va être propagée à travers la pile d'exécution jusqu'à ce qu'elle soit traitée.
— le traitement : il correspond à la capture de l'exception (utilisation des mots-clés **`try`** et **`catch`**).

1 Capturer une exception : le bloc `try ... catch`

On place dans un bloc `try` le code qui est susceptible de produire des erreurs et on capture l'exception créée avec le bloc `catch`. Lorsqu'une erreur survient dans le bloc `try`, la suite des instructions du bloc est abandonnée et le premier bloc `catch` correspondant à l'erreur est exécuté. Si on ne met rien dans le bloc `catch`, l'exception est ignorée, ce qui n'est pas une bonne pratique ! Comme plusieurs exceptions peuvent être levées par les instructions du code du bloc `try`, on peut avoir plusieurs `catch` correspondants à chacune des exceptions possibles. Attention cependant, les clauses `catch` seront testées séquentiellement. Il faut donc veiller à ne pas récuperer l'exception d'une classe et tenter de récupérer l'exception d'une classe de sa décendance. Une fois que le bloc **`catch`** est exécuté, l'exécution du code en dehors du bloc **`try`** ... **`catch`** se poursuit.

Dans l'exemple ci-dessous, on anticipe une erreur arithmétique (une division par 0). Dans ce cas, on imprime un message. Si une autre erreur intervient, on affiche la trace de la pile d'exécution.

```java
int d=10,t1=5,t2=5 ;
try{
  System.out.println("vitesse :" + d / (t2-t1)) ;
}
catch(ArithmeticException e){
   System.out.println(" vitesse non valide ") ;
}
catch(Exception e){
   e.printStackTrace() ;
}
```

On peut faire suivre les clauses `catch` par une clause `finally` qui sera *toujours* exécutée, que l'exécution se soit passée sans problèmes ou non. Ce bloc est facultatif.

2 Déléguer la capture d'une exception : `throws`

Le bloc `try` ... `catch` permet de capturer une exception. On peut aussi choisir de déléguer la capture de l'exception à la méthode appelante (elle-même pouvant faire de même de façon récursive). On a donc le schéma suivant :

```java
principale() {
   ...
   try{
     ...
     aux() ;
     ...
   }
   catch(ExceptionType e){
     // traitement exception
   }
}
```

```java
aux() throws ExceptionType {
   ...
   // appel à une méthode susceptible de lever une
   // exception de type ExceptionType
   // ou bien
   if(condition_erreur)
     throw new ExceptionType(args) ;
}
```

Ainsi, après la détection de l'erreur, on a donc bien besoin d'une phase de propagation pour remonter à la méthode qui va effectivement traiter l'exception. Evidemment, il faut qu'un méthode se charge de traiter l'exception et le compilateur peut vérifier si une telle méthode existe : cette méthode doit traiter précisemment le type d'exception ou bien une exception parente de l'exception qui a été levée. JAVA nous offre donc un choix de traiter finement les exceptions, ou bien de les traiter plus globalement. On peut par exemple imaginer que la méthode `main` contient un bloc `try` ... `catch(Exception e)`. Comme toutes les exceptions que le programmeur peut traiter héritent de la classe `Exception`, on aura fait un traitement général de toutes les erreurs pouvant survenir ! On a ainsi le choix d'avoir un traitement assez précis ou plutôt général de l'exception.

3 Créer sa propre Exception

Pour lever une exception, il suffit d'utiliser le mot-clé `throw` suivi d'un objet dont la classe dérive de la classe `Throwable`. Il faut alors l'indiquer dans la méthode qui contient l'instruction `throw` en ajoutant à la déclaration de la méthode le mot clé `throws` suivi du nom de la classe qui gère l'exception. Cette

indication à une valeur documentaire, mais aussi aide le compilateur à vérifier que l'exception est prise en compte à chaque fois que cette méthode est appelée.

```
1 public class PotionMagiqueException extends Exception {
2     public PotionMagiqueException(){
3       super() ;
4    }
5     public PotionMagiqueException(String s){
6       super(s) ;
7    }
8 }
```

```
1 public class GourdePotionMagique {
2    private int quantite, gorgee=2,contenance=20 ;
3    public GourdePotionMagique(){ quantite=0 ;}
4
5    public boolean bois() throws PotionMagiqueException {
6      if (quantite-gorgee <0)
7        throw new PotionMagiqueException(" pas assez de potion magique!") ;
8    }
9 }
```

Chapitre 7

Entré et sortie, fichiers, sauvegarde

On appelle entrée/sortie tout échange de données entre le programme et une source :

- entrée : au clavier, lecture d'un fichier, communication réseau
- sortie : sur la console, écriture d'un fichier, envoi sur le réseau

L'orientation objet du langage permet de regrouper des opérations similaires. JAVA utilise la notion de *flux* (*stream* en anglais) pour abstraire toutes ses opérations. Ici, le flux est un flux de données d'une source vers une destination. Le flux a donc une *direction*, on parle toujours soit d'un flux en entrée (qui pars d'une source et qui arrive dans le programme pour être traité) soit d'un flux en sortie (données traitées par le programme et qui sont envoyées à une destination). La nature de la source et de la destination peuvent être très différentes : un fichier, la console, le réseau. Le nature des données qui transitent dans le flux peut aussi varier : on distinguera principalement un flux de `bytes` (une suite de huit `0` ou `1`) ou un flux de caractères. La Figure 7.1 représente une partie de la hiérarchie de classes qui gère ces entrées/-sorties. La classe `File` contient des outils pour accéder aux informations d'un fichier, on en parlera plus tard. On peut voir deux classes abstraites qui manipulent les flux en entrée (`InputStream`) et les flux en sortie (`OutputStream`). On a représenté deux implémentations de chacune de ces classes abstraites selon la nature du flux (flux venant ou sortant d'un fichier ou d'un Objet). Finalement, on observe deux classes abstraites pour lire et écrire dans les flux (`Reader` et `Writer`). On a représenté deux implémentations qui diffèrent selon la nature du flux : un flux de `bytes` peut être traité par les classes `InputStreamReader` et `OutputStreamWriter` alors qu'un flux de caractères est traité par les classes `BufferedReader` et `BufferedWriter`. En résumé on obtient :

- Direction du Flux :
  - objets qui gèrent des flux d'entrée : `in`
    => `InputStream`, `FileInputStream`, `FileInputStream`
  - objets qui gèrent des flux de sortie : `out`
    => `OutputStream`, `FileOutputStream`, `FileOutputStream`
- Source du flux :
  - **fichiers** : on pourra avoir des flux vers ou à partir de fichiers
    => `FileInputStream` et `FileOutputStream`
  - **objets** : on pourra envoyer/recevoir un objet via un flux
    => `ObjectInputStream` et `ObjectOutputStream`

On peut considérer que les classes pour lire et écrire sont des outils pour transformer les flux. Un

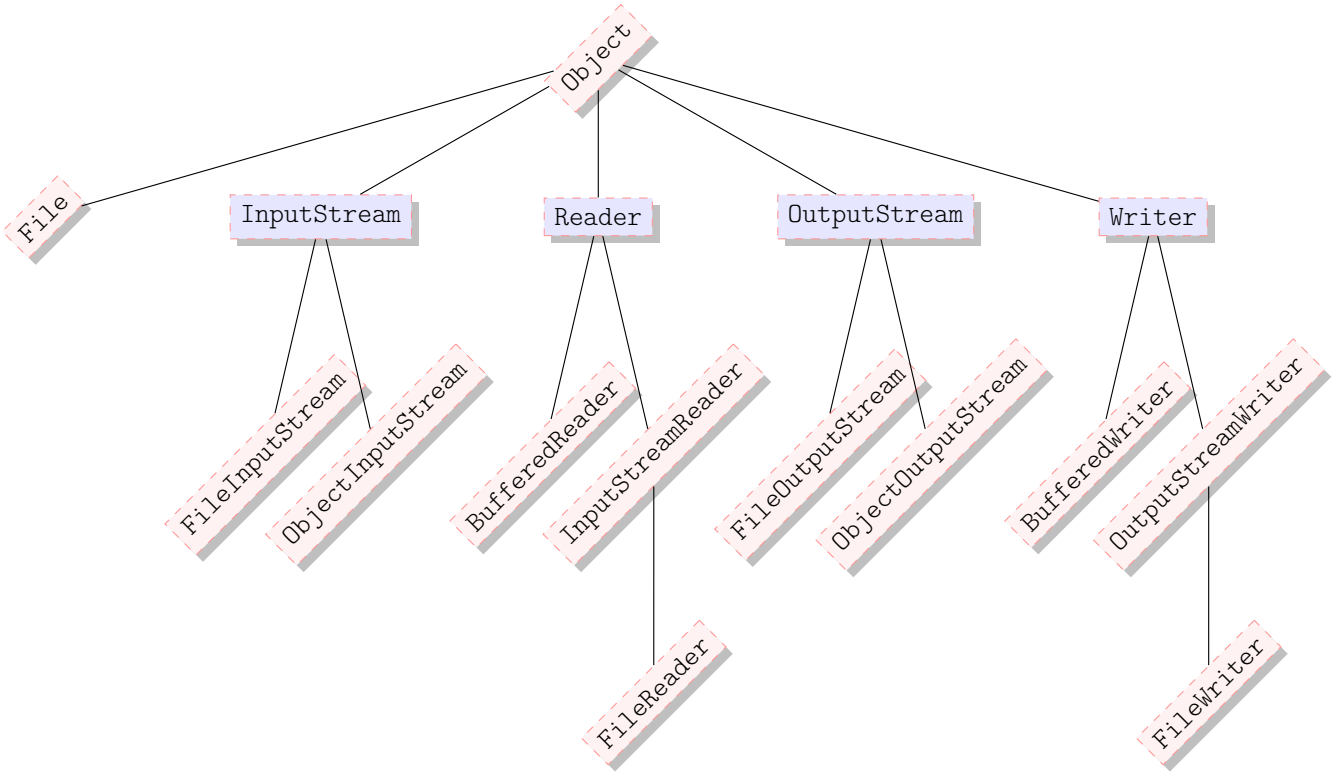


FIGURE 7.1 – Fragment de la hiérarchie de classes pour les entrées/sorties

`InputStreamReader` prend en entrée un flux de `bytes` et le transforme en un flux de caractères `char`. De manière inverse, le `OutputStreamWriter` transforme un flux de caractères en flus de `bytes`. Lorsqu'on a un flux de `char` en entrée, on peut alors utiliser la classe `BufferedReader` pour lire, via une mémoire tampon, ce qui se trouve dans le flux. De même pour la sortie, on peut écrire en sortie dans `BufferedWriter` qui utiliser une mémoire tampon et générer le flux de `char` en sortie. La Figure 7.2 résume ces étapes.

Notez bien que les opérations de lecture et d'écriture peuvent échouer pour des raisons multiples. Pour certaines, on peut avoir un contrôle (par exemple le fichier n'existe pas, on n'a pas d'accès en lecture ou écriture dans le répertoire), d'autres sont en dehors de tout contrôle (mauvais secteur du disque). Ainsi, toutes les opérations de lecture ou d'écriture sont susceptibles de lever des exceptions, qu'il faudra capturer. Il existe un hiérarchie d'exceptions dédiées aux erreurs d'entrée/sortie : ces exceptions héritent de la classe `IOException`.

1 Lire depuis la console, afficher sur la console

Pour lire, la console peut être accédée via ce que l'on appelle l'entrée « standard »`System.in`, qui est un objet de type `InputStream`. La classe `Scanner` est une classe très utile pour récupérer ce qui est tapé dans la console, en particulier récupérer et traduire automatiquement des entiers `int`, des nombres à virgule (`float` ou `double`), des chaînes de caractères (`String`).

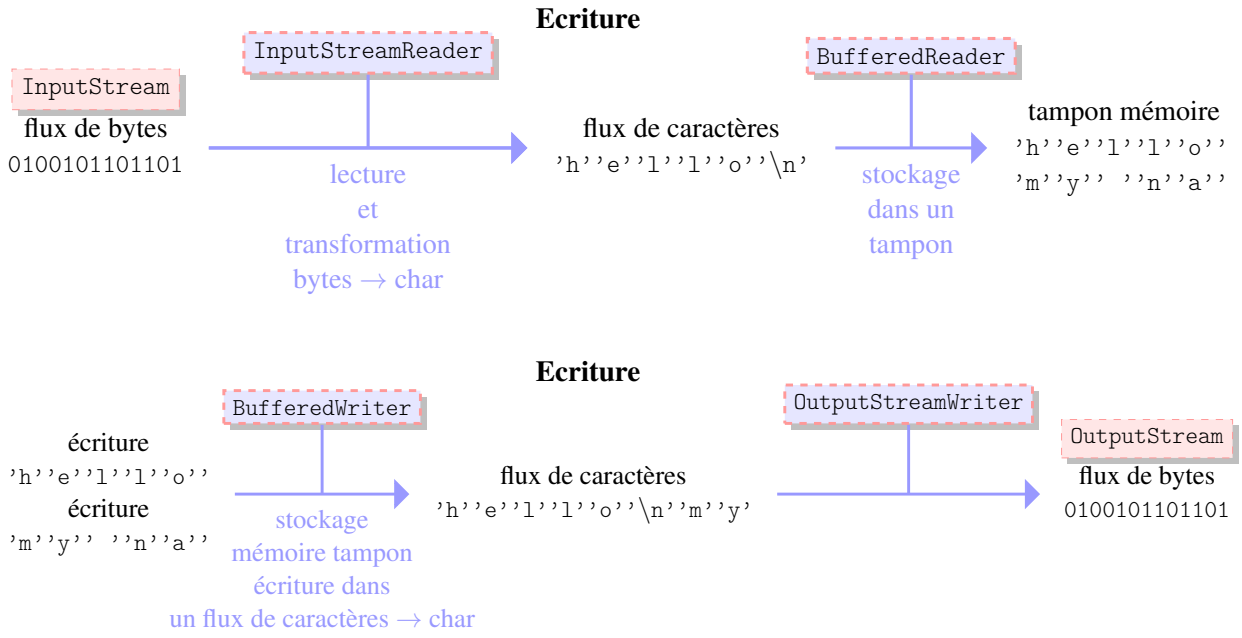


FIGURE 7.2 – Encapsulation BufferedReader(InputStreamReader) et BufferedWriter(OuputStreamWriter)

```
1 Scanner scan = new Scanner(System.in) ;
2 int n = scan.nextInt() ;
3 double x = scan.nextDouble() ;
4 String s = scan.nextLine() ;
```

Pour l'écriture, on accède la console via ce qui est appelé la « sortie standard » `System.out`, qui est de type `PrintStream` (lui-même héritant de `OutputStream`). Donc la fameuse expression `System.out.println( a_string)` ; est simplement l'écriture d'une chaîne de caractère dans un flux de `char` vers la sortie standard.

2 Lecture/Ecriture d'un fichier

La classe `File` permet d'obtenir des informations diverses sur les fichiers, on donne des exemples dans la Table 7.1. C'est aussi grâce à cette classe qu'on peut lire le contenu d'un fichier ou accéder à ce fichier pour écrire. Un fichier est une suite de `0` et `1` enregistrée sur le disque. Ainsi, pour écrire ou lire dans un fichier, JAVA utilise un flux de `bytes`. Pour écrire ou lire un fichier avec des `char`, on utilisera un objet `BufferedReader` ou `BufferedWriter`.

— En lecture : la classe `FileReader` va lire le fichier et placer le contenu dans un flux de `bytes`.
— En écriture : la classe `FileWriter` va prendre prendre un flux de bytes en entrée et écrire le flux dans le fichier

/

Dans l'exemple suivant, on va lire le premier octet d'un fichier (i.e., les 8 premières `bytes`) qui se nomme `ex.txt`. On instancie un objet de type `File` qui va accéder au fichier `ex.txt`. On va créer un flux en lecture à partir de cet objet. On aurait pu faire cela en deux étapes, mais ici, on le fait en une

— nom, chemin absolu, répertoire parent
— s'il existe un fichier d'un nom donné en paramètre
— droit : l'utilisateur a-t-il le droit de lire ou d'écrire dans le fichier
— la nature de l'objet (fichier, répertoire)
— la taille du fichier
— obtenir la liste des fichiers
— effacer un fichier
— créer un répertoire
— accéder au fichier pour le lire ou l'écrire

TABLE 7.1 – Information accessible depuis la classe `File`

seule étape en encapsulant l'instance de `File`. Ensuite, on lit le flux de `bytes` à l'aide de la méthode `read`. Evidemment, on obtient des `bytes` que l'on range dans un tableau. Une fois la lecture effectuée, on peut traiter l'information, ici, on se contente simplement d'afficher à l'écran la chaîne de caractère qui correspond à cet octet. On n'oublie bien sûr pas de fermer le flux avec la méthode `close`.

```
1 FileInputStream fis =
2                 new FileInputStream(new File(" ex.txt")) ;
3 byte[] huitLettres = new byte[8] ;
4 int nbLettreLues = fis.read(huitLettres) ;
5 for(int i=0 ;i<8 ;i++)
6    System.out.println(Byte.toString(huitLettres[i])) ;
7 fis.close() ;
```

Notez que dans ce code, on n'a pas tenu compte des exceptions :

— l'instanciation de l'objet `FileInputStream` peut lever une exception de type `FileNotFoundException`
— l'appel à la méthode `read` peut lever une exception de type `IOException`.
— la fermeture du flux avec la méthode `close` peut aussi lever une exception de typeIOException.

Il faudrait donc modifier ce code pour prendre en compte ces exceptions (soit en utilisant un bloc `try` ... `catch` soit en délégant la capture de l'exception à une méthode appelante, voir Section 6).

Dans l'exemple suivant, on affiche le contenu d'un fichier sur la console. On accède au fichier en instanciant un objet de type `File`, on utilise un flux de byte pour lire le fichier avec un objet `FileReader`, et on utilise un objet de type `BufferedReader` pour transformer le flux de `bytes` en flux de `chars`. On lit le fichier ligne par ligne en utilisant la méthode `readLine()`. De nouveau, cet exemple ne prend pas en compte les exceptions.

```
1 BufferedReader reader =
2    new BufferedReader(new FileReader(new File("ex.txt"))) ;
3 String line = reader.readLine() ;
4 while(line != null){
5    System.out.println(line) ;
6    line = reader.readLine() ;
7 }
8 reader.close() ;
```

Ci-dessous on combine les deux exemples en traitant les exceptions avec un bloc `try` . . . `catch`.

```
try {
   FileInputStream fis = new FileInputStream(new File("test.txt")) ;
   byte[] buf = new byte[8] ;
   int nbRead = fis.read(buf) ;
   System.out.println("nb bytes read : " + nbRead) ;
   for (int i=0 ;i<8 ;i++)
     System.out.println(Byte.toString(buf[i])) ;
   fis.close() ;

   BufferedReader reader =
       new BufferedReader(new FileReader(new File("test.txt"))) ;
   String line = reader.readLine() ;
   while (line!= null){
     System.out.println(line) ;
     line = reader.readLine() ;
   }
   reader.close() ;
} catch (FileNotFoundException e) {
   e.printStackTrace() ;
}
catch(IOException e){
   e.printStackTrace() ;
}
```

3 Lecture/Ecriture d'un objet : la sérialisation

Le mécanisme de « sérialisation »permet de stocker et transmettre une instance de classe. Pour se faire, la classe en question doit implémenter l'interface `Serializable`. D'une façon surprenante, cette interface *n'*a *pas* de méthode. On peut considérer qu'elle n'a qu'un rôle de *marqueur* pour indiquer que cet objet peut être sérialisé. C'est JAVA qui se charge de traduire l'objet dans un format qui n'est pas lisible par le programmeur. Pour se faire :

— en lecture : un objet `ObjectOutputStream` traduit l'objet sérialisable en un flux de `bytes`.
— en écriture : un objet `ObjectInputStream` traduit un flux de `bytes` dans un objet.

Dans l'expemple qui suit, on va sérialiser un objet de type `IrreductibleGaulois` pour l'écrire dans un fichier et le lire par la suite. Pour la l'écriture, on utilise un `ObjectOutputStream` et on appelle la méthode `writeObject()` pour traduire l'objet. Pour la lecture, on utilise `ObjectInputStream` et sa méthod `readObject`. Notez que lors de la lecture, le compilateur ne peut savoir le type de l'objet qui est lu. C'est pour cela qu'on utilise un cast explicite, le programmeur sachant que l'objet est bien du type `IrreductibleGaulois`.

```
1  IrreductibleGaulois panoramix =
2          new IrreductibleGaulois("Panoramix", 1.75) ;
3
4  ObjectOutputStream oos =
5          new ObjectOutputStream(
6            new FileOutputStream(
7              new File("panoramix.txt"))) ;
8
9  oos.writeObject(panoramix) ;
10 oos.close() ;
11
12 ObjectInputStream ois =
13         new ObjectInputStream(
14           new FileInputStream(
15             new File("panoramix.txt"))) ;
16
17 IrreductibleGaulois copyPanoramix =
18         (IrreductibleGaulois) ois.readObject() ;
19 System.out.println(copyPanoramix.nom) ;
20
21 ois.close() ;
```

Notez que dans cet exemple, le code n'est pas correct car il manque la gestion des exceptions.

Evidemment, pour qu'une classe soit « sérialisable », il serait souhaitable que tous les objets qui la composent soit sérialisable. Or, il se peut qu'un attribut ne le soit pas. Dans ce cas, on peut utiliser le mot clé `transient` pour indiquer de ne pas enregistrer cet attribut.