LU2IN002 - Introduction à la programmation orientée-objet

Christophe Marsala

SCIENCES SORBONNE UNIVERSITÉ

Cours 10 – 2 décembre 2022

Programme du jour

1. Fiabilité du code
2. Flux d'E/S

Plan du cours

Fiabilité = respect des règles de développement

Idée

Pour éviter les erreurs, respecter les règles :

- choix des noms pour comprendre qui fait quoi
- classes et méthodes de taille raisonnable, limiter les accès public
  - les opérations complexes sont déléguées à d'autres classes
  - le client voit peu de choses : facile à comprendre, évite les failles
  - taille limitée = on peut envisager de relire le code de la classe si nécessaire
- évolutivité/architecture réfléchie (pour éviter les modifications ultérieures...), usage de final (cf. prochain cours)...

$\Longrightarrow$ Plus le code est clair, plus les erreurs sont faciles à voir

Vérifications statiques

Idée

Vérifier un maximum de chose au niveau de la compilation...
$\Longrightarrow$ plus facile à corriger

- Par défaut le compilateur vérifie
  - syntaxe (les ;, parenthèses, accolades...)
  - type des variables, compatibilité avec les instances et méthodes
  - niveau d'accès (aux méthodes, variables...)
- D'autres propriétés sont plus difficiles à montrer et nécessitent plus d'informations transmises au compilateur
  - langage d'annotations

Annotations standards

Certaines erreurs ne posent pas de problème de compilation mais provoquent des comportements étranges lors de l'exécution... Ce sont les plus chères à corriger !

- Exemple : on veut redéfinir toString dans une classe :

```
1 public class Point {
2     ...
3     public String toStrig() {
4         return "Point [x=" + x + ", y=" + y + "]";
5     }
6 }
```

- et on écrit ailleurs :

```
1 public class TestPoint {
2     public static void main(String[] args) {
3         Point a = new Point();
4         System.out.println("Mon point : "+a.toString());
5     }
6 }
```

- Quel affichage ?
- Pas d'erreur MAIS problème lors de l'exécution !

Annotations standards

Certaines erreurs ne posent pas de problème de compilation mais provoquent des comportements étranges lors de l'exécution... Ce sont les plus chères à corriger !
Les annotations permettent d'en prévenir certaines.

Par exemple : @Override qui signale une redéfinition de méthode

```
1    @Override
2    public String toStrig() {
3        return "Point [x=" + x + ",y=" + y + "]";
4    }
```

Provoque une erreur de compilation :

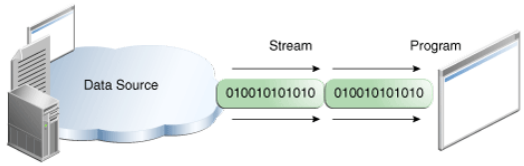
```
1 Point.java:23: method does not override or implement a method
2                from a supertype
3 @Override
4 ^
5 1 error
```


Plan du cours

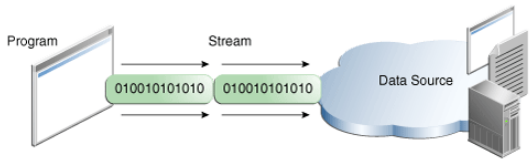
Entrées/Sorties d'un programme

Les entrées et sorties sont gérées séparément par flux

IN :



OUT :



En Java = panoplie d'outils pour communiquer dans les deux sens avec toutes sortes de sources de données

Entrées et sorties de base

Définition

Un flux (=stream) : entrées/sorties (dynamiques) d'un programme

Il s'agit d'outils essentiels pour :
- lire/écrire des fichiers
- sauver les paramètres d'un programme dans un fichier
- communiquer avec d'autres postes de travail (en réseau)
- travailler à plusieurs sur un projet (=partager des fichiers)
- lire/écrire dans des bases de données (BD)
- saisir des informations depuis le clavier
- écrire sur la console (le terminal)

Schéma général : l'exemple des fichiers (1)

Pour accéder à un fichier en lecture
1. Vérifier l'accès : existence du fichier, possibilité de lecture,...
2. Ouvrir un flux en lecture depuis le fichier
3. Lire dans le fichier
4. Fermer le flux ouvert

- Remarque : beaucoup de choses se gèrent comme les fichiers
  - clavier, réseau, bases de données,...

Schéma général : l'exemple des fichiers (2)

Pour accéder à un fichier en écriture
1. Vérifier l'accès : possibilité de création, d'écriture...
2. Ouvrir un flux en écriture vers le fichier
   - entraîne sa création le cas échéant
3. Écrire dans le fichier
4. Fermer le flux ouvert

- Remarque : beaucoup de choses se gèrent comme les fichiers
  - clavier, réseau, bases de données,...

Représentation des fichiers

La classe File

Cette classe permet de créer un objet représentant un fichier

- test d'existence
- distinction fichier/répertoire
- copie/effacement
- ...

- boolean canExecute()
- boolean canRead()
- boolean canWrite()
- boolean delete()
- boolean isDirectory()
- boolean isFile()
- File[] listFiles()
- boolean mkdir()

Nombreuses opérations très intéressantes concernant la manipulation des fichiers

Lecture de fichiers

1. File : pour représenter un fichier par un objet
2. FileInputStream : création d'un flux en lecture depuis fichier
   - il y a des **exceptions** à gérer
   - il faut penser à **fermer** les fichiers ouverts

```
1  FileInputStream in = null;
2  File f = new File("tatouine.txt");
3  try {
4      in = new FileInputStream(f); // ouverture du fichier
5      // throws: FileNotFoundException : => try/catch
6
7      // OPERATIONS DE LECTURE
8
9  } catch (FileNotFoundException e) {
10     // Instructions pour gérer l'exception...
11
12 } finally {
13     if (in != null) {
14         in.close();
15     }
16 }
```


Les petits pièges... La fermeture des fichiers

1. Toujours fermer un fichier ouvert...

```
1 try { FileInputStream in =
2           new FileInputStream( new File(filename) );
3         ... // LECTURE
4         in.close();
5 catch(...){...}
```

2. Même s'il y a des erreurs pendant la lecture !

```
1 try { FileInputStream in =
2           new FileInputStream( new File(filename) );
3       ... // LECTURE
4       in.close();
5 catch(...){ in.close(); }
```

3. Mais ça ne compile pas !

```
1 FileInputStream in = null;
2 try { in = new FileInputStream(new File(filename));
3       ... // LECTURE
4       in.close();
5 catch(...){   in.close(); }
```

4. Plus élégant : lignes 4-5 ⇒ finally{in.close();}

Les petits pièges... La fermeture des fichiers

5. La solution précédente ne marche pas encore !

```
1 FileInputStream in = null;
2 try { in = new FileInputStream(new File(filename));
3       ... // LECTURE
4 finally{  in.close(); }
```

6. ... le close est susceptible de lever une exception si le fichier n'est pas ouvert (NullPointerException) !

```
1  FileInputStream in = null;
2  File f = new File("tatouine.txt");
3   try {
4      in = new FileInputStream(f); // ouverture du fichier
5      // Throws: FileNotFoundException : => try/catch
6
7      // OPERATIONS DE LECTURE
8
9      }
10  } finally {          // On est sûr de passer par là
11     if (in != null) { // vérifier que le fichier est ouvert
12         in.close();
13     }
14  }
```


Lecture de fichiers (byte stream)

public int read() throws IOException

Reads a byte of data from this input stream. This method blocks if no input is yet available.
Returns :
the next byte of data, or -1 if the end of the file is reached.
Throws :
IOException - if an I/O error occurs.

Exemple :

```
1 int c = in.read(); // lecture d'un octet d'information
2         // susceptible de lever IOException => try/catch
3 while (c  != -1) {  // tant que fin de fichier non atteinte
4     System.out.print(c);     // affichage
5     c = in.read();     // lecture du caractère suivant
6 }
```

Ce qui peut aussi s'écrire plus simplement :

```
1  while ((c = in.read()) != -1) {
2     System.out.print(c);
3   }
```


Limites de FileInputStream

- Lecture octet par octet
  - accès bas niveau au fichier
- Rappels Java : représentation interne
  - entiers : binaire (fort/faible) signé en complément à 2
    - byte : 1 octet
    - short : 2 octets
    - int : 4 octets
  - réels : norme IEEE 754, signe+exposant+significande
    - float : 4 octets
    - double : 8 octets
  - char : 2 octets (Unicode)
- Bilan : complexe de reconstruire des valeurs lues par octet

Limites → Nouvelles classes

- Des classes supplémentaires enrichissent les FileStream :
- Classe de lecture de haut niveau : DataInputStream

```
DataInputStream istream = null;
try {
    // on encapsule le flux dans un objet DataInputStream
    istream = new DataInputStream(
      new FileInputStream(new File("tatouine.dat")));

    System.out.println(istream.readChar());
    System.out.println(istream.readDouble());
    System.out.println(istream.readInt());
    System.out.println(istream.readChar());
    // mais pas de fonctions pour les String...
}catch (.... ) {
    ... // à compléter...
} finally{
    if (istream != null)
        istream.close();
}
```

- Il existe une classe équivalente pour les flux de sortie

Ecriture de fichiers (byte stream)

public FileOutputStream(String name) throws FileNotFoundException

Creates an output file stream to write to the file with the specified name.
Parameters :
name - the system-dependent filename
Throws :
FileNotFoundException - if the file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason

- Fonction proche de celle d'ouverture en lecture... avec une option supplémentaire : ajouter des valeurs dans un fichier...
- public FileOutputStream(String name, boolean append) throws FileNotFoundException

Ecriture de fichiers

- Exemple d'utilisation (Oracle Java Tutorials) :

```
FileInputStream in = null;
FileOutputStream out = null;

try {
    in = new FileInputStream("tatouine.txt");
    out = new FileOutputStream("dagobah.txt");
    int c = in.read();

    while (c  != -1) {
        out.write(c);
        c = in.read();
    }
}catch (.... ) {
    ...
} finally {
    if (in != null) {
        in.close();
    }
    if (out != null) {
        out.close();
    }
}
```

- Que fait ce programme ?