Conception et Programmation Objet - GM3

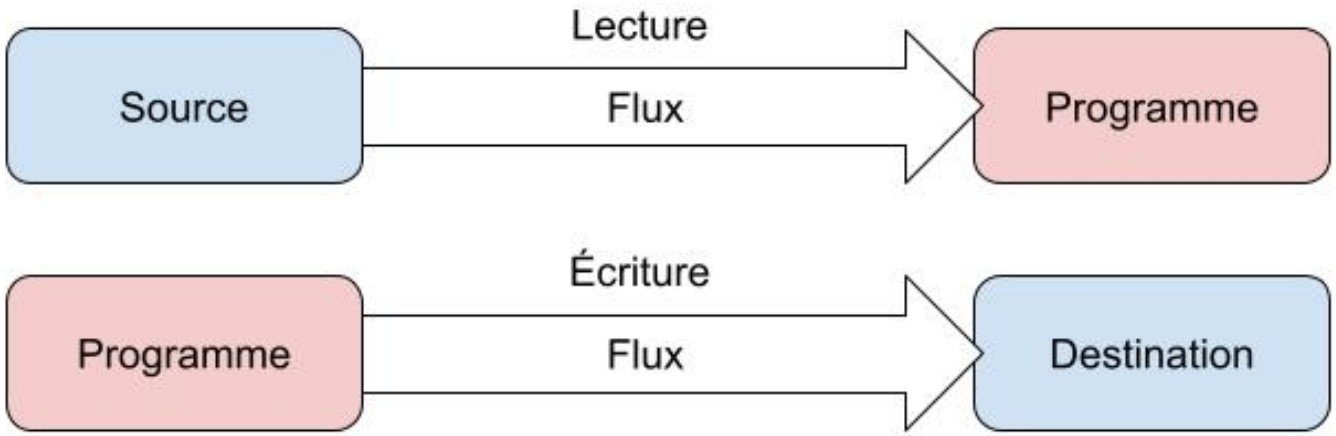
Entrées/Sorties, packages et compilation

Mathieu Bourgais

2023

Plan

Principe des flux



- En Java, le transport de données se fait à travers des **flux** unidirectionnels
- Il faut donc commencer par **ouvrir** un flux, en lecture ou en écriture, puis **traiter les données** dans ces flux
- Ces flux sont indépendants des données (même s'ils peuvent être redéfinis pour être plus précis) -> Un ensemble commun de méthodes de lecture ou d'écriture

Les deux types de flux

- Java propose deux types de flux généraux : Les flux **orientés octets** et les flux **orientés caractères**
- Les flux orientés octets permettent de manipuler des informations binaires sur 8 bits (peut lire des sons ou des images par exemple)
- Les flux orientés caractères permettent de manipuler des chaînes des caractères sur 16 bits (lire/écrire du texte)
- Chacun de ces flux définit des classes pour **lire** et pour **écrire** des données
- Il existe aussi des flux "bufferisés" orientés octets ou orientés caractères utiles pour travailler sur des fichiers
- L'ensemble des classes se trouve dans **Java.io**

Les classes importantes en lecture d'octets

- **InputStream** est la classe mère de toutes les autres
- Pour manipuler des fichiers, on peut utiliser la classe **FileInputStream** (classe "conteneur")
- Pour manipuler les données, on peut ensuite utiliser les classes **BufferedInputStream**, **ObjectInputStream**, **DataInputStream** (classes de "traitement")

```
import Java.io.BufferedInputStream;
import Java.io.FileInputStream;

BufferedInputStream is = new BufferedInputStream(
    new FileInputStream("adresse/monFichier"));
```

Les classes importantes en écriture d'octets

- **OutputStream** est la classe mère de toutes les autres
- Pour manipuler des fichiers, on peut utiliser la classe **FileOuputStream** (classe "conteneur")
- Pour manipuler les données, on peut ensuite utiliser les classes **BufferedOutputStream**, **ObjectOutputStream**, **DataOutnputStream** (classes de "traitement")

```
import Java.io.BufferedOutputStream;
import Java.io.FileOutputStream;

BufferedOutputStream os = new BufferedOutputStream
    (new FileOuputStream("adresse/monFichier"));
```

Les méthodes importantes

- Pour la lecture, on dispose de la méthode **int read()** et de quelques dérivés pour plus de précision
- Pour l'écriture, on dispose de la méthode **int write(int i)** et de quelques dérivés pour plus de précision
- La méthode **close()** permet de fermer un flux après son utilisation (et de libérer les ressources mémoires)
- Les autres méthodes sont à retrouver dans la documentation

Exemple de manipulation de flux d'octets

```
DataOutputStream dos = new DataOutputStream(new
    FileOutputStream("monFichier"));

dos.writeBoolean(true);
dos.write(12);
dos.close();

DataInputStream is = new DataInputStream(new
    BufferedInputStream(new FileInputStream("
    monFichier")));

System.out.println(is.readBoolean());
System.out.println(is.readInt());
is.close();
```

Les flux standards et Scanner

- Il existe par défaut des flux standards qui communiquent avec le terminal
- **System.in** est le flux standard de lecture
- **System.out** est le flux standard d'écriture
- Lorsqu'on fait *System.out.print("test")* on utilise la méthode *print()* du flux standard de sortie
- *InputStream* ne générant que des octets, la classe **Scanner** a été développée pour inclure nativement des méthodes de lecture plus pratiques
- **ATTENTION** : La fermeture du *Scanner* ferme aussi le flux associé (bien souvent *System.in*) ! Une bonne pratique consiste à ouvrir le Scanner dans le *main* et à le passer en paramètre aux méthodes qui vont en avoir besoin puis à fermer le Scanner avant de terminer le *main*.

Les classes importantes en lecture de caractères

- **Reader** est la classe mère de toutes les autres
- Pour manipuler des fichiers, on peut utiliser la classe **FileReader** (classe "conteneur")
- Pour manipuler les données, on peut ensuite utiliser les classes **BufferedReader**, **InputStreamReader** (classes de "traitement")

```
import Java.io.BufferedReader;
import Java.io.FileReader;

BufferedReader monReader = new BufferedReader(new
    FileReader("adresse/monFichier.txt"));
```

Les classes importantes en écriture de caractères

- **Writer** est la classe mère de toutes les autres
- Pour manipuler des fichiers, on peut utiliser la classe **FileWriter** (classe "conteneur")
- Pour manipuler les données, on peut ensuite utiliser les classes **BufferedWriter**, **OutputStreamWriter**, **PrintWriter** (classes de "traitement")

```
import Java.io.BufferedWriter;
import Java.io.FileWriter;

BufferedWriter monWriter = new BufferedWriter(new
    FileWriter("adresse/monFichier.txt"));
```

Les méthodes importantes avec les caractères

- Pour la lecture, on dispose de la méthode **int read()** et de quelques dérivés pour plus de précision
- **String readLine()** dans *BufferedReader* permet de lire une ligne d'un coup
- Pour l'écriture, on dispose de la méthode **void write(int i)** et de quelques dérivés pour plus de précision
- **void newLine()** dans *BufferedWriter* permet d'insérer un caractère de fin de ligne
- La méthode **close()** permet de fermer un flux près son utilisation (et de libérer les ressources mémoires)
- Les autres méthodes sont à retrouver dans la documentation

Exemple de manipulation de flux de caractères

```
BufferedReader monReader = new BufferedReader(new
    FileReader("adresse/monFichier1.txt"));

String s = monReader.readLine();

BufferedWriter monWriter = new BufferedWriter(new
    FileWriter("adresse/monFichier2.txt"));

monWriter.write(s);
monWriter.newLine();

monReader.close();
monWriter.close();
```

Concept de la sérialisation

- En plus d'envoyer des informations de types simples dans les flux, on peut vouloir envoyer des instances d'objets (notamment pour les enregistrer dans des fichiers)
- Pour cela, il faut d'abord "transformer" l'objet de sorte à ce qu'il puisse être écrit/lu depuis un flux -> Sérialisation
- La sérialisation s'effectue simplement en implémentant l'interface **Serializable** dans la classe que l'on souhaite écrire, et dans toutes les classes qui la compose (et qu'on souhaite écrire). L'interface *Serializable* ne contient pas de méthodes
- À la lecture, l'objet n'est pas reconstruit par son constructeur, il est directement récupéré comme il est enregistré

Fonctionnement de la sérialisation

- Une fois que l'objet implémente la classe *Serializable*, il pourra être passé dans un flux
- On utilisera alors les classes **ObjectInputStream** et **ObjectOutputStream**
- On dispose alors des méthodes **writeObject(Object o)** et **Object readObject()**
- On peut utiliser le mot-clé **transient** pour ne pas sérialiser un champs de l'objet
- Les attributs *static* ne sont pas non plus sérialisés

Exemple de sérialisation

```
public class Magicien extends PJ implements
    Soigneur, Serializable{
     ...
}

Magicien merlin = new Magicien();
ObjectOutputStream os = new ObjectOutputStream(
    BufferedOutputStream(new FileOuputStream("
    adresse/monFichier")));

os.writeObject(merlin);
os.close();
```

Plan

Mettre son code dans des packages

- Mettre toutes ses classes dans un même dossier peut devenir illisible dans un gros projet
- De plus, on peut avoir envie de réutiliser un ensemble de classes issues d'un projet sans réutiliser tout le projet (principe d'une bibliothèque à charger)
- Pour cela, on va placer ses classes dans des **packages**
- L'utilisation des *packages* permet de désambiguïser des classes de même nom (espace de nommage)

Utilisation des packages

- On indique que la classe appartient à un *package* en l'écrivant avant de déclarer la classe sous la forme **package nom.du.package**
- Un package est équivalent à une arborescence de dossiers : le package *nom.du.package* équivaut à un chemin **./nom/du/package**
- Deux classes qui font partie du même package se connaissent
- Pour utiliser une classe d'un autre package, il faut l'importer avec **import nom.du.package.MaClasse**
- On peut aussi importer toutes les classes d'un package avec **import nom.du.package.***
- Mieux vaut démarrer un projet avec des packages qu'en rajouter en cours de route (certains IDE sont un peu capricieux avec cela)

Options de compilation

- Le compilateur **javac** propose quelques options qui permettent d'organiser son projet avec un dossier contenant les fichiers sources et un dossier contenants les classes compilées
- L'option -**classpath** permet d'indiquer un chemin vers des classes déjà compilées qui pourraient être utiles à la compilation
- L'option -**sourcepath** permet d'ndiquer le chemin vers les fichiers sources à compiler
- L'option -**d** permet d'indiquer un chemin où stocker les classes compilées
- D'autres options sont à retrouver dans la documentation

Plan

Modifications de l'exemple

- Ajouts de packages pour mieux organiser le projet
- Ajout d'une classe **Gestionnaire** pour gérer le chargement et l'enregistrement depuis des fichiers
- On va enregistrer puis charger notre donjon donc sérialisation des classes concernées

Sérialisation de l'exemple

- **Donjon** implémente *Serializable*
- Puisque le donjon contient des salles, **Salle** implémente *Serializable*
- Puisque les salles contiennent soit des monstres, soit des médecins, **Monstre** et **Medecin** implémentent *Serializable*
- Puisque les monstres et médecins héritent des personnages, si on veut conserver toutes les informations des personnages, on doit aussi implémenter *Serializable* dans **Personnage**
- Par héritage, on retire l'implémentation de *Serializable* dans *Monstre* et *Medecin*

Lire le nom du marchand

```
public String lireNomMarchand(String nomFichier){
    BufferedReader is = null;
    String result = "␣";
    try {
        is = new BufferedReader(new FileReader(
            nomFichier));
        result = is.readLine();
        is.close();
        return result;
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
            e.printStackTrace();
    }
    return result;
}
```

Enregistrer le donjon

```
public void enregistrerDonjon(Donjon d, String
    nomFichier){
     ObjectOutputStream os = null;
     try {
          os = new ObjectOutputStream(new
              BufferedOutputStream(new
              FileOutputStream(nomFichier)));
          os.writeObject(d);
          os.close();
     } catch (IOException e) {
          e.printStackTrace();
     }
}
```

Modification du main

```
Marchand aeris = new Marchand();
Donjon monDonjon = new Donjon();
Gestionnaire gestion = new Gestionnaire();

aeris.setNom(gestion.lireNomMarchand("marchand.txt
    "));
System.out.println(aeris.sePresenter());

gestion.enregistrerDonjon(monDonjon, nomSauvegarde
    );

Donjon donjonCharge = gestion.chargerDonjon(
    nomSauvegarde);
System.out.println(donjonCharge.getMesSalles().get
    ("vert"));

scan.close();
```

Résultat de l'exécution du main

```
Bonjour, je m'appelle Aerith
Dans quel fichier enregistrer le donjon ?
monDonjon
Je suis la salle vert et je contiens nom=Docteur Quinn
Je suis la salle bleu et je contiens nom=Patrick
```

L'ensemble du code, et de l'arborescence du projet, sont à retrouver sur Moodle.