Programmation orientée objet en Java

Les entrées et sorties de données en Java

Dominique Blouin
Télécom Paris, Institut Polytechnique de Paris
dominique.blouin@telecom-paris.fr

Objectifs d’apprentissage

- Flux de données d’entrée et de sortie
- Flux de caractères
- Flux de données binaires
- L’API NIO2
- Sérialisation des objets

Tout est octet (byte)

- La mémoire RAM est composée d'**octets** (bytes).
- Les fichiers sur disque / disque SSD (Solid-State Drive) sont également composés d'**octets**.
- Quand deux ordinateurs communiquent, ils échangent des **octets**.
- Les octets sont classiquement représentés en base 16 (hexadécimal).
  - Par exemple : 0x7A.

Entrées / sorties en Java

- Les entrées-sorties (E/S, I/O en anglais) en Java sont basées sur la notion de **flux d'octets** (byte stream).
- **Flux de sortie** (output stream) dans lequel on **écrit** des octets.
- **Flux d'entrée** (input stream) à partir duquel on **lit** des octets.
- Un flux d'entrée ou de sortie peut correspondre à :
  - Un fichier sur le disque de l'ordinateur (ou un autre périphérique).
  - Une zone de mémoire.
  - Une connexion réseau (socket).
  - Etc.

La classe `InputStream`

- C’est la classe **de base** de tous les **flux d'entrée**.
  - Classe **abstraite**.

- Ses sous-classes permettent de lire des octets (bytes) sur **différents supports**.

- Recherche : JAVA SE **`InputStream`**
  - Voir les méthodes de base de la classe.
  - Voir la classe **`FileInputStream`**.
  - Voir la classe **`ByteArrayInputStream`**.

Exemple : lire le contenu d'un fichier

- On désire afficher le contenu d'un fichier, c'est à dire la suite des octets qui composent le fichier sous forme hexadécimale.

- La première chose à faire est d'utiliser une structure **`try-catch`** car presque toutes les opérations d'entrée-sortie peuvent **lever des exceptions**.

```
    try {
        ...
    }
    catch (IOException ex) {
        LOGGER.log(Level.SEVERE, "Error while ...",
ex);
    }
```

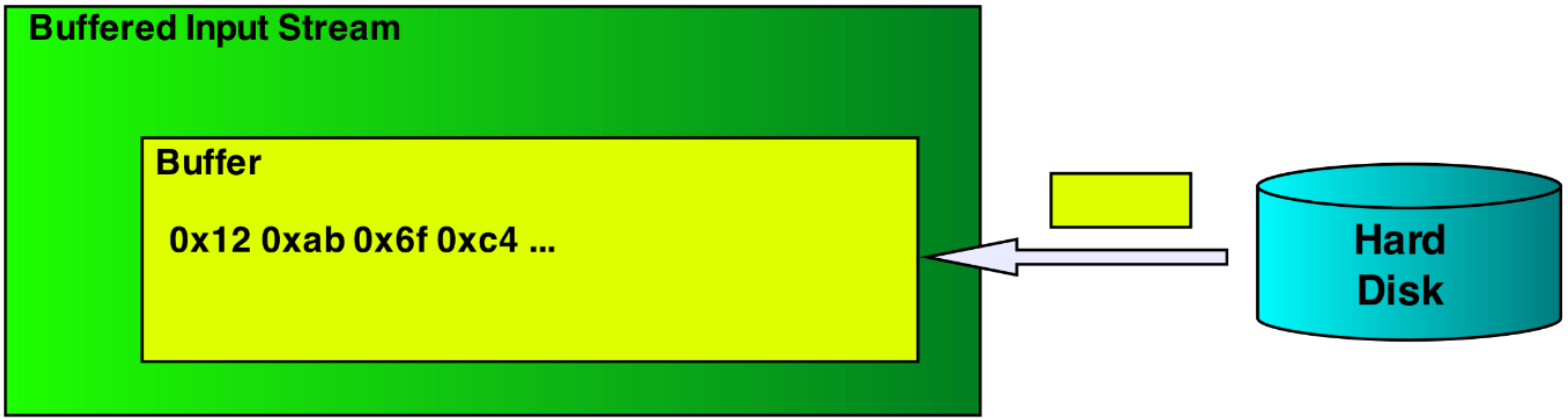
Exemple : lire le contenu d'un fichier

```
public class HexFile {

    public static void main(String[] args) {
        FileInputStream inputStream = null;

        try {
            inputStream = new FileInputStream(args[0]);
            int byteData = inputStream.read();

            while (byteData != -1) {
                String digits = Integer.toHexString(byteData);

                if (byteData < 16) {
                    digits = "0" + digits;
                }

                System.out.println(" 0x" + digits);
                byteData = inputStream.read();
            }
        }
        catch (IOException ex) {
            LOGGER.log(Level.SEVERE, "Error while reading file '" + args[0] + "'.", ex);
        }
        finally {
            try {
                if (inputStream != null) {
                    inputStream.close();
                }
            }
            catch (IOException ex) {}
        }
    }
```

Mise en mémoire tampon des E/S (buffering)

- Imaginons un programme qui lit ou écrit des octets dans un fichier sur un support matériel (disque SSD, clé USB, etc.).
- Si chaque instruction de lecture ou d'écriture provoque la mise en marche du matériel, le programme sera lent et le matériel ne fonctionnera pas longtemps...
- On règle ce problème par l'utilisation d'une zone de **mémoire intermédiaire** entre le programme et le flux appelée **tampon** (buffer en anglais).

Mise en mémoire tampon des entrées



- Quand un octet doit être lu, il est d'abord recherché dans le tampon.
- Si le tampon est vide, il sera rempli à partir du disque SSD en **une seule opération** de lecture.
  - Cela **réduit** le nombre d'opérations physiques de lecture.
- La taille du tampon est un paramètre pour lequel une valeur par défaut est fournie.

Exemple de mise en mémoire tampon des entrées

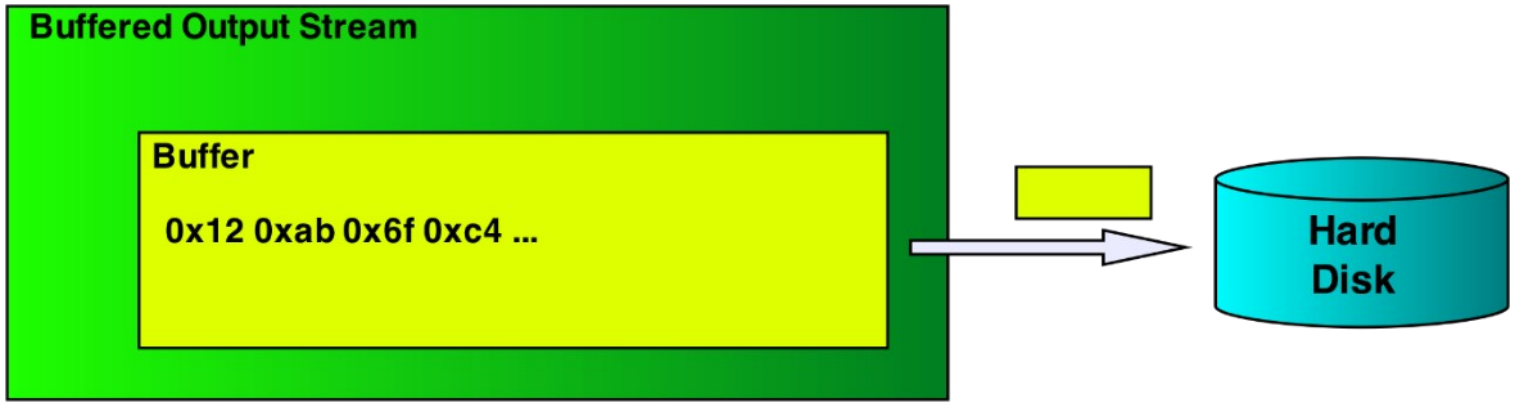
- Pour mettre en tampon un flux d'entrée d'octets, on **encapsule** ce flux dans une instance de la classe **`BufferedInputStream`**.

```
FileInputStream fileInpStr = new
FileInputStream("myDataFile.bin");
BufferedInputStream bufInpStream = new
BufferedInputStream(fileInpStr);
```

- Puis l'instance de la classe **`BufferedInputStream`** s'utilise comme une instance de **`FileInputStream`**, mais en interne, elle réalise la gestion de mise en mémoire tampon.

```
int byteData = bufInpStream.read();
```

Mise en mémoire tampon des sorties



- Quand un octet doit être écrit, il est d'abord écrit dans le tampon.
- Si le tampon est plein, son contenu sera alors copié sur le disque SSD **en une seule opération** d'écriture.
  - Cela **réduit** le nombre d'opérations physiques d'écriture.
- La taille du tampon est un est un paramètre pour lequel une valeur par défaut est fournie.

Exemple de mise en mémoire tampon des sorties

- Pour mettre en tampon un flux d'entrée d'octets, on **encapsule** ce flux dans une instance de la classe **`BufferedOutputStream`**.

```
FileOutputStream fileOutStr = new
FileOutputStream("myDataFile.bin");
BufferedOutputStream bufOutStream = new
BufferedOutputStream(fileOutStr);
```

- Puis l'instance de la classe **`BufferedOutputStream`** s'utilise comme une instance de **`FileOutputStream`**, mais en interne, elle réalise la gestion de mise en mémoire tampon.

```
bufOutStream.write(byteData);
```

Classes pour flux d'octets de *données*

- **`FileInputStream`** et **`FileOutputStream`** pour la lecture et l'écriture d'octets dans des **fichiers** sur disque.
- **`ByteArrayInputStream`** et **`ByteArrayOutputStream`** pour la lecture et l'écriture d'octets dans la **mémoire** du programme.
- Une classe par **type de support** des données.

Classes pour encodage des données

- Si tout est octet (byte), il est rare que les programmes doivent manipuler **directement des octets**.
- En revanche, toutes les données sont **codées** avec des octets.
- Deux types particuliers de flux d'entrée et de sortie :
  - Flux de **caractères**.
  - Flux de données **binaires**.

Flux de caractères

- Il existe des **jeux de caractères** (charset) et des **codages** de ces caractères (encoding).
- Jeu de caractères : liste des symboles que l'on considère comme étant des caractères.
- Codage : manière dont les caractères sont représentés sous forme d'octets.
- Un jeu de caractères contient des symboles visibles mais aussi **invisibles** comme le caractère « saut de ligne ».
- Un des plus anciens jeux de caractères est l'ASCII (American Standard Code for Information Interchange).
  - Contient 128 caractères, chacun étant codé sur 7 bits.
  - Ne convient pas pour des textes en français :
    - Ne permet pas d'encoder un nombre suffisamment grand de caractères.

Flux de caractères

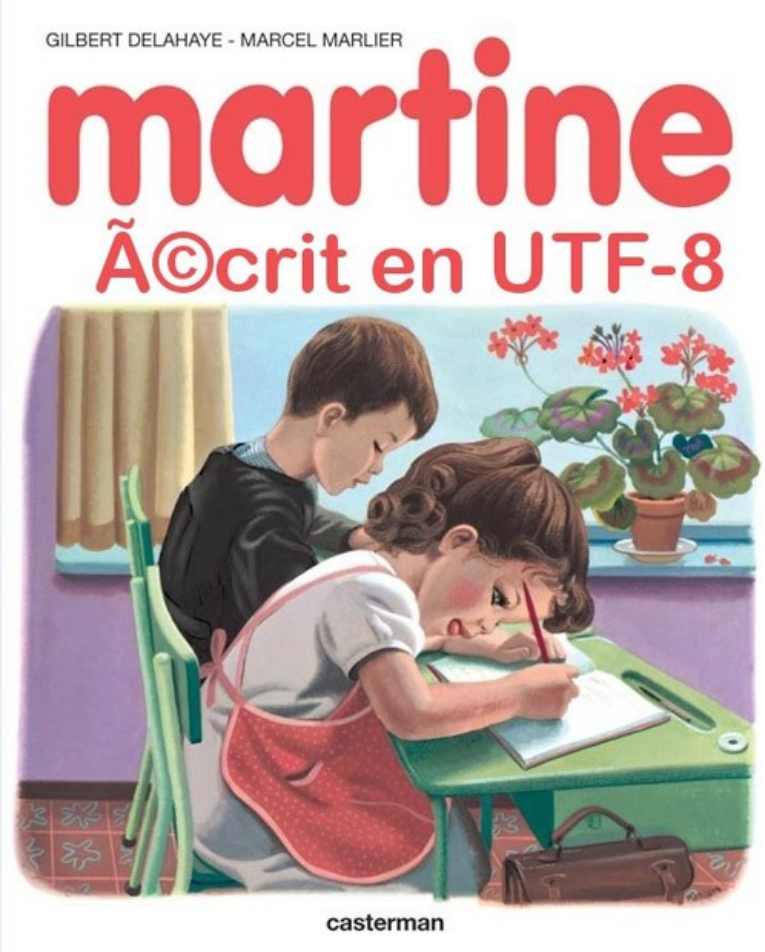
- En mémoire, Java utilise le jeu de caractères **UNICODE**, qui vise à donner à tout caractère de n'importe quel système d'écriture un **nom**.
  - L'encodage le plus courant des numéros de caractères UNICODE est défini dans la norme **UTF-8** (Universal Transformation Format).

- Il n'est pas, a priori, nécessaire de connaître les détails des jeux de caractères et de leurs codages.

- Lorsque l'on **écrit** des caractères, il convient de choisir un **jeu de caractères** et son **encodage**.
  - Utiliser UNICODE/UTF-8 est en général suffisant.

- Lorsque l'on **lit** des caractères, il est nécessaire de connaître le **jeu de caractères** et l'**encodage** utilisés lors de l'écriture.

Erreurs d’encodage



- Si l'on se trompe de jeux de caractères en lecture, des erreurs de décodage peuvent apparaître…
- On le constate parfois sur les pages Web, dans les courriels, ou dans les fenêtres de terminal par exemple.

Ecriture de caractères

- On **encapsule** un flux de sortie d'octets dans un objet qui sait **coder** les caractères en octets.

- Une de ces classes d'encapsulation est **`PrintWriter`**.
  - Recherche : JAVA SE `PrintWriter`

- Exemple : Création d'un objet **`PrintWriter`** :

```
FileOutputStream fileOutStr = new FileOutputStream("data.txt");
PrintWriter printWriter = new PrintWriter(fileOutStr);
```

- Ou bien :

```
PrintWriter printWriter = new PrintWriter(new
FileOutputStream("data.txt"));
```

- Ou bien plus simplement :

```
PrintWriter printWriter = new PrintWriter("data.txt");
```

- Notons que nous n'avons **pas** explicitement utilisé de mémoire tampon, car la classe **`PrintWriter`** le gère au niveau **caractère**.

Utilisation de la classe `PrintWriter`

```
printWriter.print("abc ");
printWriter.println(12);
printWriter.println('x');

printWriter.close();
```

- Résultat dans le fichier **`data.txt`** :

```
abc 12
x
```

Lecture de caractères

- On **encapsule** un flux d'entrée d'octets dans un objet qui sait **décoder** les octets en caractères.

- La principale classe d'encapsulation est la classe **`Scanner`**.
  - Recherche : JAVA SE Scanner

- Création de l'objet **`Scanner`** et mise en tampon :

```
FileReader fileReader = new FileReader("data.txt");
BufferedReader bufReader = new BufferedReader(fileReader);
Scanner scanner = new Scanner(bufReader);
```

Lecture de caractères

- On relit le fichier data.txt :

```
abc 12
x
```

```
String myString = scanner.next(); // myString = "abc"
int myInt = scanner.nextInt(); // myInt = 12
String nextLine = scanner.nextLine(); // nextLine = "" (no input
skipped)
String lastLine = scanner.nextLine(); // lastLine = "x"

scanner.close();
```

Flux de données binaires

- Lorsque l'on veut stocker des données de base (**`int`**, **`float`**, etc.), il est possible de les stocker sous forme textuelle, mais ce n'est pas très efficace.
  - Idem si l'on veut transférer des données par un **réseau**.

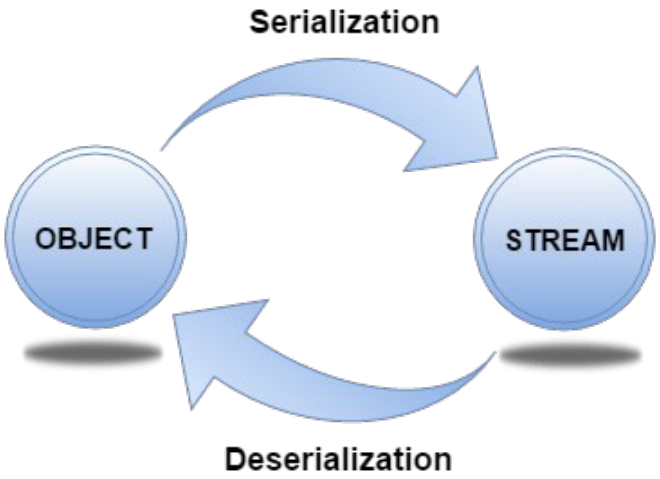
- Tous ces types de base ont une représentation bien plus compacte que la représentation textuelle ; c'est leur représentation **binaire** en mémoire.

- Pour stocker des données sous format binaire, Java fournit des classes qui englobent des flux d'octets et y écrivent ou y lisent les représentations binaires des types de base.
  - Recherche : JAVA SE **`DataInputStream`**
  - Recherche : JAVA SE **`DataOutputStream`**

Exemples d'écriture et lecture de données en binaire

```
FileOutputStream fileOutStr = new FileOutputStream("data.bin");
BufferedOutputStream bufOutStr = new BufferedOutputStream(fileOutStr);
DataOutputStream dataOutStr = new DataOutputStream(bufOutStr);
dataOutStr.writeInt(123);
dataOutStr.writeLong(234L);
dataOutStr.writeString("abc");
dataOutStr.writeFloat(12.3f);

FileInputStream fileInpStr = new FileInputStream("data.bin");
BufferedInputStream bufInpStr = new BufferedInputStream(fileInpStr);
DataInputStream dataInpStr = new DataInputStream(bufInpStr);
int myInt = dataInpStr.readInt();
long myLong = dataInpStr.readLong();
String myString = dataInpStr.readString();
float myfloat = dataInpStr.readFloat();
```

Sérialisation des objets



- De même qu'il est possible d'écrire la représentation binaire d'une **donnée de base** dans un flux d'octets, il est également possible d'écrire la représentation binaire d'un **objet**.
- **Sérialiser** un objet consiste à calculer sa **représentation binaire**.
  - C'est le processus de **sérialisation**.
- Lors de la sérialisation d'un objet, tous les objets **référencés par ses attributs** sont également **sérialisés** avec lui.
- Donc si une structure de données est sérialisée, une liste par exemple, tous les **éléments qu'elle contient** seront sérialisés avec elle.
- **Désérialiser** un objet consiste à convertir un flux de données binaires en objet.
  - C'est l'**inverse** de la sérialisation

Tout n’est pas sérialisable…

- Par exemple, un objet de type **`FileInputStream`** n'est **pas sérialisable**.
- En règle générale, tout ce qui **référence** quelque chose **à l'extérieur de la machine virtuelle** n'est **pas sérialisable**.
- De même, un objet relatif à l'exécution particulière d'un programme comme un processus léger (**`Thread`**) n'est pas sérialisable.
- Techniquement, ce sont uniquement des « données pures » que l'on voudra sérialiser.

Spécifier qu'un objet est sérialisable

- C'est au programmeur de spécifier si une classe d'objet est sérialisable.
- Cela se fait en déclarant que cette classe implémente l'interface **`java.io.Serializable`**.
- Cette interface ne contient rien (interface de **marquage**) ; il ne s'agit que d'une indication pour le compilateur.
- La classe devra contenir un numéro de série (ou de version) :

```
public static final long serialVersionUID = 202406181720L;
```

- Cet identifiant est composé de la manière suivante :
  - <Année><Mois><Jour><Heure><Minutes>
- Une classe peut évoluer. Si un objet est sérialisé avec une **version donnée** de sa classe, il ne peut être relu qu'avec la **même version** de la classe.

Exemple de classe sérialisable

```
public class Person implements Serializable {

    public static final long serialVersionUID = 202406181830L;

    private final String firstName;

    private final String lastName;

    public Person(String firstName,
                  String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    ...
}
```

- Ne pas oublier de **modifier** la valeur de **`serialVersionUID`** lorsque les declarations d'attributs de la classe sont **modifiées**.

Sérialiser un objet

- Recherche : JAVA SE **ObjectOutputStream**

```
Person person = new Person("Joe","Bleau");

OutputStream fileOutStream = new FileOutputStream("Joe_Bleau.bin");
OutputStream bufOutStream = new
BufferedOutputStream(fileOutStream);

ObjectOutputStream objOutStream = new
ObjectOutputStream(bufOutStream);
objOutStream.writeObject(person);

objOutStream.close();
```

Que faire des objets non sérialisables, ou qui ne sont pas à sérialiser ?

```
public class Person implements Serializable {

    public static final long serialVersionUID = 202406181830L;

    private final Set<Observer> observers; // No need to serialize them

    private final String firstName;

    private final String lastName;

    public Person(String firstName,
                  String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;

        observers = new HashSet<>();
    }

    ...
}
```

Utiliser le mot clé *transient*

```
public class Person implements Serializable {

    public static final long serialVersionUID = 202406181830L;

    private final transient Set<Observer> observers; // Will not be serialized

    private final String firstName;

    private final String lastName;

    public Person(String firstName,
                  String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;

        observers = new HashSet<>();
    }

    ...
}
```

- Le mot clé **`transient`** signifie que les valeurs de l'attribut ne seront **pas sérialisées**.

Appel des constructeurs lors de la désérialisation

- Lors de la désérialisation des objets (construction des objets à partir des données sérialisées precédemment), il n'est pas possible de déterminer quel constructeur de la classe à instancier doit être appelé.
- Ainsi, **aucun des constructeurs** ne sera appelé, pas même le constructeur ne prenant aucun paramètre, s'il en existe un.
- Ainsi, un attribut déclaré **transient**, tel que l'ensemble des observateurs du modèle, ne sera pas initialisé, ce qui peut poser un problème…

Solution : utiliser l'instanciation *paresseuse*

```
public class Person implements Serializable {

    public static final long serialVersionUID = 202406181830L;

    private final transient Set<Observer> observers; // Will not be serialized

    private final String firstName;

    private final String lastName;

    public Person(String firstName,
                  String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    protected Set<Observer> getObservers() {
        if (observers == null) {
            observers = new HashSet<>();
        }

        return observers
    }
    ...
}
```

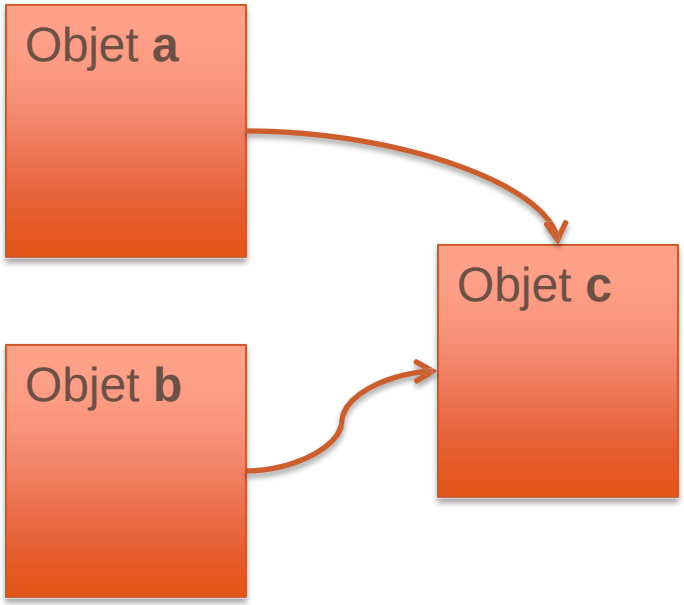
Constructeur non utilisé lors de la désérialisation

Stratégie d'instanciation paresseuse (lazy) via encapsulation des données

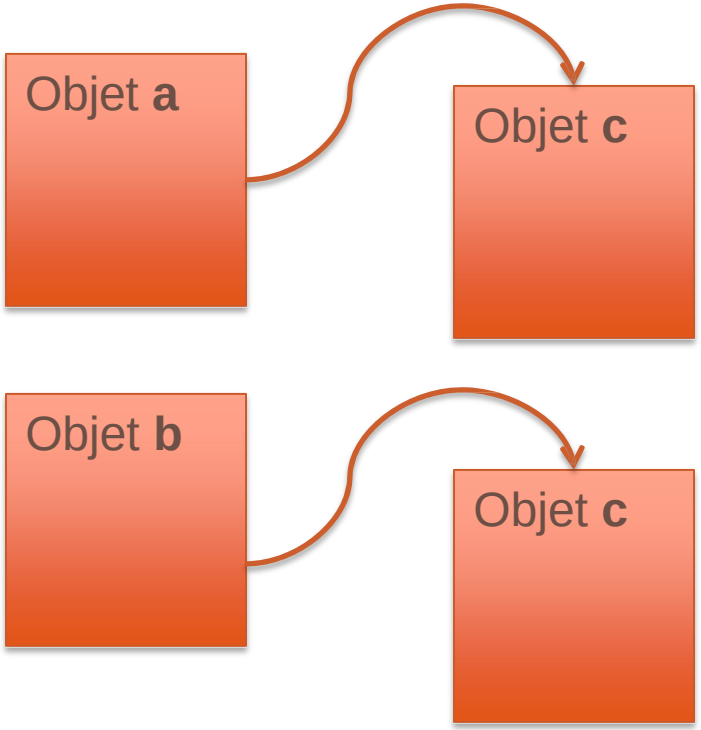
Attention !

- Il faut éviter d'écrire **plusieurs fois le même objet**, même après modification, car Java gère cela avec un système de cache assez compliqué.
- Cela peut créer un problème lorsque des objets sont référencés par **plusieurs objets**.
- Une solution : créer un objet **x** qui référence **a** et **b**, puis sauvegarder cet objet **x** plutôt que **a** et **b** séparément.

Exemple de deux objets qui réfèrent un même objet

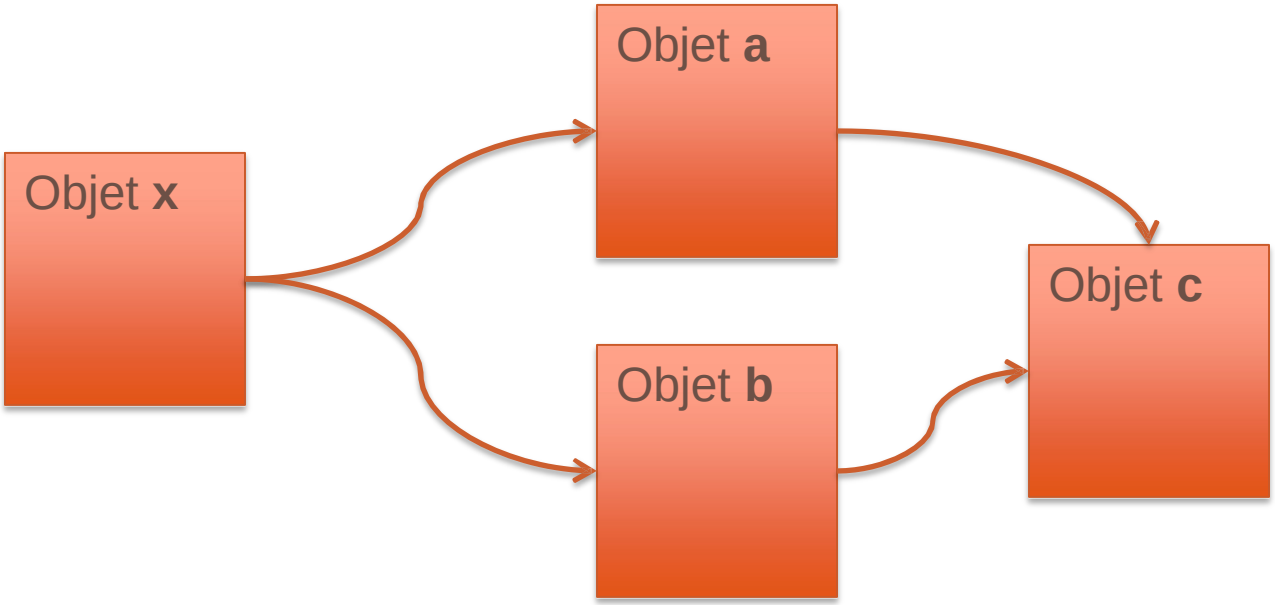


Sauvegarde de **a**
puis
sauvegarde de **b**



Lecture de **a**
puis
lecture de **b**

Solution : utiliser un seul objet racine



Sauvegarde de **x**
puis
lecture de **x**

Avantages et désavantages de la sérialisation des données

- Manière facile d'implémenter la sauvegarde des objets.
- Mais pas très efficace…
- Problème lors des changements de version :
  - Rétrocompatibilité.
  - Maintenance de numéros de version.
- Problèmes de **sécurité**.
  - https://www.youtube.com/watch?v=uur5B0rFMkQ

Fermeture des flux

- L'un des problèmes des entrées-sorties de Java est que tous les flux doivent être **fermés** après leurs utilisation.
  - Sinon, le système d'exploitation fermera éventuellement ces flux, mais peut-être pas tout de suite.
- Or il y a un **nombre limité** de flux que l'on peut ouvrir.
- Cela peut poser des problèmes...

Fermeture des flux: solution classique

```
FileOutputStream fileOutStr = null;
BufferedOutputStream bufOutStr = null;
DataOutputStream dataOutStr = null;

try {
    fileOutStr = new FileOutputStream("data.bin");
    bufOutStr = new BufferedOutputStream(fileOutStr);
    dataOutStr = new DataOutputStream(bufOutStr);
    ...
    dataOutStr.writeInt(123);
    ...
}
catch (IOException ex) {
    LOGGER.log(Level.SEVERE, "Error opening file 'data.bin'.", ex);
}
finally {
    try {
        fileOutStr.close();
        bufOutStr.close();
        dataOutStr.close();
    }
    catch (Exception ex) {
        ...
    }
}
```

- Question : lesquels de ces flux doivent être fermés?

Fermeture des flux : solution `try-with-resource`

- Nouvelle forme de **`try-catch`** appellée **`try-with-ressources`**.
  - La fermeture des flux est gérée **automatiquement**.

```
try (
    FileOutputStream fileOutStr = new FileOutputStream("data.bin");
    BufferedOutputStream bufOutStr = new
BufferedOutputStream(fileOutStr);
    DataOutputStream dataOutStr = new DataOutputStream(bufOutStr);
) {
    ...
    dataOutStr.writeInt(123);
    ...
}
catch (IOException ex) {
    LOGGER.log(Level.SEVERE, "Error opening file 'data.bin'.", ex);
}
```

New Input Output (NIO)

- Il existe plusieurs manières **d'emboîter** les classes permettant d'utiliser des flux de données en entrée ou en sortie.
  - Cela permet de couvrir un **grand nombre** de cas d'utilisation.

- Ce trop-plein de possibilités peut être perçu comme un **inconvénient** du système d'I/O de Java.

- De plus, l'API demeure relativement **compliquée**.
  - Par exemple, dans l'exemple de la planche précédente, il faut instancier **trois objets** avant de commencer à écrire des données binaires.

- Il serait donc utile de pouvoir encapsuler un code tel que celui de la planche précédente dans des méthodes plus simples afin d'éviter de le dupliquer.

- Pour solutionner ce problème, Java a introduit **NIO** (New Input Output), puis une version améliorée de **NIO** (**NIO2**) à partir du JDK 7.

NIO2

- NIO2 revoit complètement les entrées sorties pour les systèmes de fichiers, et introduit un **modèle** de système de fichiers.

- Ce modèle contient les classes :
  - **`Path`** : qui représente un chemin dans un système de fichiers.
  - **`Files`** : qui est une classe utilitaire contenant des méthodes statiques pour manipuler les éléments d'un système de fichiers.
  - **`FileSystemProvider`** : qui est un fournisseur de services pour interagir avec un système de fichiers sous-jacent.
  - **`FileSystem`** : qui représente un système de fichiers.
  - **`FileSystems`** : qui permet de **fabriquer** une instance de **`FileSystem`** (factory).

- Recherche : JAVA SE **`Files`**

Exemple d'entrées-sorties fichier avec NIO2

- Lecture de fichier texte avec l'API classique :

```
try (
    FileReader fileReader = new FileReader("data.txt");
    BufferedReader bufReader = new BufferedReader(fileReader);
) {
    List<String> lines = new ArrayList<String>();
    String line = bufReader.readLine();

    while (line != null) {
        lines.add(line);
        line = bufReader.readLine();
    }
}
catch (IOException ex) {
    ...
}
```

- Lecture de fichier texte avec NIO2 :

```
try {
    List<String> lines = Files.readAllLines(Paths.get("data.txt"),
        StandardCharsets.UTF_8);
}
catch (IOException ex) {
    ...
}
```

D’autres flux

- Il existe beaucoup d'autres types de flux fournis avec le JDK ou disponibles dans des bibliothèques open source.
- **`GZipInputStream`** and **`GZipOutputStream`** pour la compression des flux d'octets.
- **`PipedInputStream`** and **`PipedOutputStream`** pour la communication entre des processus légers (threads).
- **`SequenceInputStream`** pour la concaténation de flux d'octets.
- **`AudioInputStream`** pour lire des fichiers audios.
- Etc.

Conclusion

- Les entrées / sorties de Java sont très riches et permettent de couvrir un très **grand nombre** de **types de données** et de **type de support** de données.

- Certaines classes sont plus simples que d'autres et il faut savoir choisir la bonne classe :
  - Ainsi, si l'on désire lire un flux de caractères uniquement **ligne par ligne**, on utilisera la classe **`BufferedReader`** qui est plus simple que la classe **`Scanner`**.

- Dans le cas où le flux correspond à un élément matériel (fichier sur disque SSD, carte réseau, etc.), on veillera à ce que le flux soit **bufferisé**.
  - Il peut être bufferisé au niveau **caractère** ou au niveau **octet**.

- Pour les caractères, il est possible de spécifier le jeu de caractères (et son codage).
  - Si non spécifié, le codage par défaut du **système d'exploitation** est utilisé.