Entrées-sorties

Université Française d'Egypte
Version O 7.2 – 1/11/13
Richard Grin

Plan

- Gestion des fichiers ; **`Path`** et **`Files`**
- Les flots (*streams*), modèle de conception « décorateur »
- Classe **`URL`**
- Noms de fichiers, ressources
- Sérialisation
- Mise en forme
- Isoler les entrées-sorties
- Expressions régulières

Gestion des fichiers
Classes **`Path`** et **`Files`**

Introduction

- Nouvelle API **NIO.2** introduite par le JDK 7, contenue dans le paquetage **`java.nio.file`**

Fonctionnalités

- Manipulation des noms de fichier
- Copier, déplacer, supprimer des fichiers
- Lister les fichiers d'un répertoire
- Afficher et modifier les métadonnées (autorisations, propriétaire,…) sur les fichiers
- Traverser une arborescence de fichiers
- Surveiller les changements dans un répertoire
- Lire et écrire le contenu de petits fichiers

Interface de base : **`Path`**

- Correspond à un nom de fichier relatif ou absolu dans un système de fichiers
- **Indépendant de l'existence ou non d'un fichier qui a ce nom**

Création d'un `Path`

- `Path path = Paths.get("/rep/truc");`
- `Path path = Paths.get("/rep", "truc");`
- Si on a un chemin relatif par rapport à un répertoire dont on a déjà un **`Path`**, il est plus souple d'utiliser la méthode **`resolve`** :

```
path = pathRep.resolve(pathRelatif);
```

- Si on veut partir du répertoire parent de **`path2`** :

```
path = path2.resolveSibling(pathRelatif);
```


Informations sur un `Path`

	/home/dupond/fich	dupond/fich
getFileName()	fich	fich
getName(0)	home	dupond
getNameCount()	3	2
subpath(0,2)	home/dupond	dupond/fich
getParent()	/home/dupond	dupond
getRoot()	/	null

- Pour Windows C:\home\dupond\fichier, getRoot renvoie C:\

Conditions sur le nom de fichier

- **`boolean startsWith`** ; paramètre **`String`** ou **`Path`**
- **`boolean endsWith`** ; paramètre **`String`** ou **`Path`**

Conversion en chemin absolu

- Utilise le répertoire courant
- **`Path toAbsolutePath()`**
  Le fichier peut exister ou ne pas exister
- **`Path toRealPath()`**
  Lance **`IOException`** si le fichier n'existe pas

Quelques propriétés système

- **`user.dir`** : répertoire courant
- **`file.separator`** : séparateur dans les noms de fichier (/ pour Unix, \ pour Windows)
- **`user.home`** : répertoire « home »
- Exemple :
  `System.getProperty("user.dir")`

Opérations sur les fichiers

- La classe **`Files`** fournit des méthodes **`static`** pour lire, écrire et manipuler (copier, déplacer, supprimer) des fichiers ordinaires et des répertoires
- La plupart des méthodes de **`Files`** peuvent lancer une **`IOException`**
- Tiennent compte de l'existence des fichiers
- Exemple :
  `Files.delete(path);`

A noter

- Les fichiers de cette section désignent les fichiers par leur chemin d'accès (**`Path`**)
- Souvent les fichiers sont considérés comme des ressources désignées par un **`URL`** et pas par leur chemin de type **`Path`**
- On verra comment travailler avec un fichier désigné par un **`URL`**, dans les sections « Classes **`URL`** et **`URI`** » et « Noms de fichiers, ressources »

Vérifications sur les fichiers

- Existence : **`exists(Path)`** et **`notExists(Path)`**
- Autorisations : **`isReadable(Path)`**, **`isWritable(Path)`**, **`isExecutable(Path)`**
- Si 2 chemins représentent le même fichier : **`isSameFile(Path, Path)`**

Gestion des fichiers

- Méthodes **`copy`**, **`move`**, **`delete`**, **`deleteIfExist`** pour copier, déplacer et supprimer un fichier
- Méthodes **`createFile`**, **`createDirectory`**, **`createDirectories`** pour créer un fichier ou un répertoire

Glob

- Pour filtrer les noms de fichiers
- Correspond aux expressions qu'on peut rencontrer dans les commandes Unix comme « **`ls l*`** »

Exemples de Glob

- **`*`** : de 0 à n caractères
- **`**`** : traverse les répertoires
- **`?`** : un seul caractère
- **`[abx]`** : un des caractères entre crochets
- **`[a-g]`** : un des caractères compris entre les extrémités

Liste des fichiers d'un répertoire

- Les fichiers d'un répertoire sont fournis sous la forme d'un flot : **`DirectoryStream<Path>`**
- Cette classe implémente **`Iterable<Path>`**, ce qui simplifie son utilisation

Exemple avec glob

```
Path rep = ...;
try (DirectoryStream<Path> flot =
       Files.newDirectoryStream(
         rep, "*.{class,jar}")) {
  for (Path fich : flot) {
    System.out.println(fich.getFileName());
  }
} catch (IOException x) {
  ...
}
```


Visiter une arborescence de fichiers

- **`Files`** contient 2 méthodes **`walkFileTree`** pour parcourir une arborescence de fichiers, en lançant des actions sur les répertoires ou les fichiers ordinaires rencontrés
- **`FileVisitor<T>`** définit les actions exécutées pendant le parcours

Exemple (1/2)

```
Path repertoire = Paths.get(…);
FileVisitor<Path> visiteur =
  new Finder("*.class");
Files.walkFileTree(repertoire, visiteur);
```

```
class Finder extends SimpleFileVisitor<Path>{
  private PathMatcher matcher;
  public Finder(String modele) {
    matcher = FileSystems.getDefault()
         .getPathMatcher("glob:" + modele);
  }
```


Exemple (2/2)

```
  @Override
  public FileVisitResult visitFile(
    Path path,
    BasicFileAttributes attributs)
  throws IOException {
    if (matcher.matches(path.getFileName())){
      System.out.println(path);
    }
    return FileVisitResult.CONTINUE;
  }
}
```


Surveiller un répertoire

- Il peut être intéressant d'être prévenu si un répertoire est modifié (fichier ajouté, modifié ou supprimé)
- Par exemple, une application peut utiliser des plugins sous la forme de fichiers jar qui sont déposés dans un répertoire
- L'interface **`WatchService`** sert à surveiller un répertoire

Lire et écrire des petits fichiers

- Des méthodes de **`Files`** permettent de lire ou d'écrire le contenu d'un fichier *en une fois*, en prenant en charge l'*ouverture et la fermeture des fichiers*
- Très pratiques pour lire ou écrire les petits fichiers
- Ne peuvent être utilisées pour les très gros fichiers, car le contenu du fichier doit être enregistré dans la mémoire centrale

2 types de fichiers

- L'API pour lire et écrire le contenu des fichiers distingue 2 types de fichier :
  - fichiers contenant des octets « bruts »
  - fichiers contenant du texte : des caractères, codés avec un certain codage (*charset* ; ASCII, UTF-8, ISO-8859-1,...)

Lire des petits fichiers

- `byte[] readAllBytes(Path) throws IOException`
  renvoie tous les *octets* d'un fichier
- `List<String> readAllLines( Path, Charset) throws IOException`
  renvoie toutes les lignes d'un fichier *texte*

Écrire des petits fichiers

- `Path write(Path, byte[], OpenOption…)`
  écrit tous les *octets* du tableau dans un fichier ; `OpenOption` indique ce qu'il faut faire si le fichier existe déjà
- `Path write(Path, Iterable<? extends CharSequence> lignes, CharSet, OpenOption…)`
  écrit les lignes de *texte* dans un fichier

Interface `CharSequence`

- Suite de `char`
- Implémentée par `String` et `StringBuilder`
- Interface utilisée pour les signatures des méthodes qui travaillent sur des suites de caractères
- Méthodes `charAt`, `length` et `subSequence` (pour extraire une sous-suite)

Options pour écrire

- Par défaut un fichier existant sera écrasé
- Pour ajouter à la fin du fichier :

```
Files.write(path, bytes,
  StandardOpenOption.APPEND);
```


Pour les plus gros fichiers

- Nous allons voir comment lire ou écrire dans un fichier d'une manière plus souple (mais plus complexe), avec des méthodes de la classe `Files`

Fichiers texte

- **`newBufferedReader(Path, Charset)`** renvoie un **`BufferedReader`** qui permet de lire un fichier ligne à ligne
- **`newBufferedWriter(Path, Charset, OpenOption…)`** renvoie un **`BufferedWriter`** qui permet d'écrire dans un fichier

Fichiers d'octets

- **`newInputStream(Path, OpenOption…)`** renvoie un **`InputStream`** pour lire dans un fichier d'octets
- **`newOutputStream(Path, OpenOption…)`** renvoie un **`OutputStream`** pour écrire dans un fichier d'octets
- Ces classes n'utilisent pas de buffer et elles sont souvent décorées avec un **`BufferedInputStream`** ou un **`BufferedOutputStream`**

Flots (*streams*)

Flots (*streams*) - définition

- Les flots de données permettent d'échanger de données entre un programme et l'extérieur

Flot - utilisation

- Lecture séquentielle d'un flot de données :
  1) Ouvrir le flot
  2) Tant qu'il y a des données à lire, lire la donnée suivante dans le flot
  3) Fermer le flot

Sources et destinations de flots

- Fichier
- *Pipe* entre 2 processus
- *Socket* pour échanger des données sur un réseau
- URL
- etc...

Paquetage `java.io`

Paquetage `java.io`

- Contient la plupart des classes liées aux flots
- Prend en compte :
  - 2 types de flots (caractères et octets)
  - différentes sources et destinations
  - « décorations » diverses

2 types de flots

- Flots de caractères pour lire ou écrire des caractères lisibles par un homme, codés (ISO 8859-1, UTF 8,…)
- Flots d'octets pour lire ou écrire des octets « bruts » manipulés par un programme

2 types de classes

- Classes de base, associées à une source ou une destination
  Exemple : **`FileReader`** pour lire un fichier
- Classes qui décorent une autre classe
  Exemple : **`BufferedReader`** qui ajoute un *buffer* pour lire un flot de caractères

Décorations des flots

- Fonctionnalités de base d'un flot : lecture ou écriture (méthodes **`read`** ou **`write`**)
- On peut lui ajouter d'autres fonctionnalités :
  - *Buffer* pour réduire les lectures ou écritures « réelles »
  - Codage/décodage des données manipulées
  - Compression/décompression de ces données
  - …

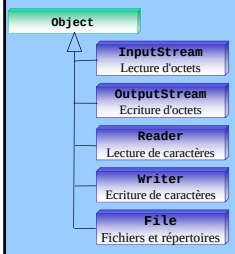
Exemple de décoration

```
FileReader fr = new FileReader(fichier);
// br décore fr avec un buffer
BufferedReader br = new BufferedReader(fr);
int c; // code Unicode du caractère lu
try {
  while ((c = br.read()) != -1)
    . . .
}
```

Fin du fichier

Grâce au buffer la plupart des **`br.read()`** n'entraînent pas une lecture réelle sur le disque

Classes de base de `java.io`



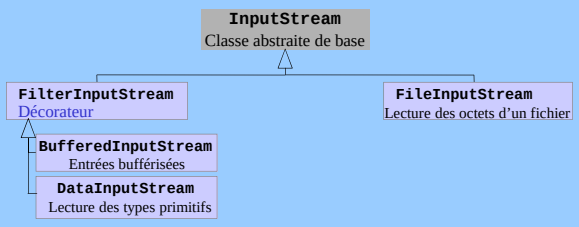
Lecture / écriture de flots d'octets

Lecture / écriture de flots de caractères

Représente fichiers et répertoires

Lecture d'un flot d'octets

Quelques classes associées à la lecture d'un flot d'octets



Méthodes de la classe `InputStream`

Ajouter `throws IOException` aux méthodes, sauf les 2 dernières

```
abstract int read()
int read(byte[] b)
int read(byte[] b, int début, int nb)
long skip(long n)
int available()
void close()

synchronized void reset()
synchronized void mark(int nbOctetsLimite)
boolean markSupported()
```


Méthode read

- **`int read()`**
  renvoie l'octet lu (compris entre 0 et 255),
  ou -1 si elle a rencontré la fin du flot
- Bloque
  - jusqu'à la lecture d'un octet
  - ou la rencontre de la fin du flot
  - ou si une exception est levée

Modèle de conception (*design pattern*) « décorateur »

Principe

- Un objet « décorateur » ajoute une fonctionnalité à un objet décoré
- Le constructeur du décorateur prend en paramètre l'objet qu'il décore
- Quand un décorateur reçoit un message, il remplit sa fonctionnalité (la « décoration »)

  Si besoin est, il fait appel à l'objet décoré pour remplir les fonctionnalités de base

Exemple

- **`isb`**, un **`InputStreamBuffer`** ajoute un buffer (disons de 512 octets) à un **`InputStream is`**
- **`isb.read()`**

  va chercher un octet dans le buffer rempli par une précédente lecture « réelle »

  Si le buffer est vide, **`isb`** demande d'abord à **`is`** de remplir le buffer (avec 512 octets)

On peut décorer un décorateur

- Un décorateur peut décorer un autre décorateur
- C'est possible parce que, selon le pattern décorateur,
  - le décorateur décore un **`InputStream`**
  - le décorateur et le décoré « *sont-des* » **`InputStream`** (par héritage)

Lire des types primitifs depuis un fichier

```
FileInputStream fis =
    new FileInputStream("fichier");
BufferedInputStream bis =
    new BufferedInputStream(fis);
DataInputStream dis =
    new DataInputStream(bis);
double d = dis.readDouble();
String s = dis.readUTF();
int i = dis.readInt();
dis.close();
```

décore **`fis`** avec un buffer

décore **`bis`** en décodant les types primitifs

Codage UTF-8 pour les **`String`**

Variante de l'exemple

- En fait, on n'a pas besoin des variables intermédiaires et on écrira :

```
DataInputStream dis =
  new DataInputStream(
    new BufferedInputStream(
      new FileInputStream("fichier"));
```

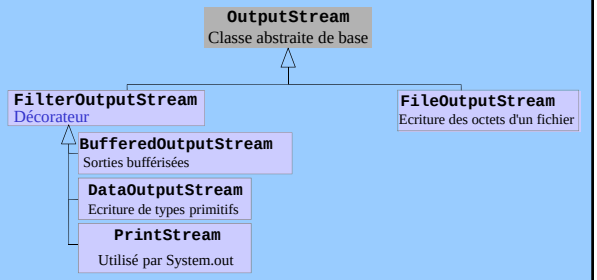
Ce code se trouvera le plus souvent dans un **`try`** avec ressource pour que les flots soient fermés automatiquement

Intérêt du pattern décorateur

- Utile quand un objet de base peut être décoré de multiples façons
- Si on utilisait l'héritage, on aurait de nombreuses classes, représentant l'objet de base, décoré d'une ou plusieurs décorations
- Avec ce pattern, on a seulement une classe par type de décoration
- De plus on peut décorer un objet dynamiquement pendant l'exécution

Écriture d'un flot d'octets

Quelques classes associées à l'écriture d'un flot d'octets



Classe `OutputStream`

`throws IOException` pour toutes les méthodes

```
abstract void write(int b)
void write(byte[] b)
void write(byte[] b, int début, int nb)
void flush()
void close()
```


Écrire des types primitifs dans un fichier

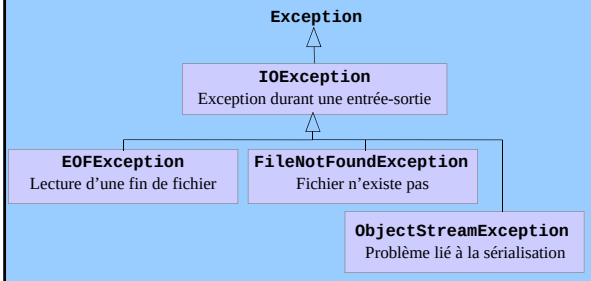
```
DataOutputStream dos =
  new DataOutputStream(
    new BufferedOutputStream(
      new FileOutputStream("fichier")));
dos.writeDouble(12.5);
dos.writeUTF("Dupond");
dos.writeInt(1254);
dos.close();
```


Écrire des types primitifs *à la fin* d'un fichier

- Le constructeur
  `FileOutputStream(String nom, boolean append)`
  permet d'ajouter à la fin du fichier
- Sinon, le contenu du fichier est effacé à la création du flot

Exceptions

Principales exceptions liées aux entrées-sorties



Traitement des exceptions ; JDK 7

```
try (
   DataInputStream dis =
    new DataInputStream(new FileInputStream("fich")) {
  while (true) {
    double d = dis.readDouble();
    . . .
  }
}
catch(EOFException e) {}
catch(FileNotFoundException e) { . . }
catch(IOException e) { . . . }
```

Vide ; juste pour sortir de la boucle **while**

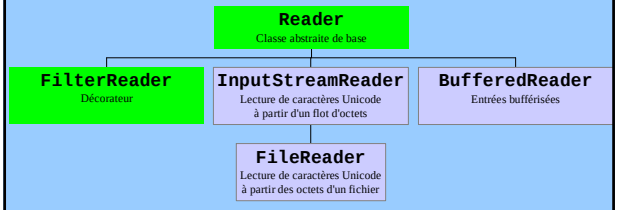
Cas où les exceptions ne sont pas traitées par la méthode – JDK 7

```
public void lire(String fichier) throws IOException {
  try (
     DataInputStream dis =
       new DataInputStream(new FileInputStream(fich))) {
    while (true) {
      double d = dis.readDouble();
      . . .
    }
  }
  catch(EOFException e) {}
}
```

Peut-on enlever cette ligne ?

Lecture d'un flot de caractères

Hiérarchie des principales classes de lecture d'un flot de caractères



Lecture d'un flot composé de lignes de texte

- On utilise la classe **`BufferedReader`** qui comprend la méthode **`readLine()`** pour lire une ligne de texte

Séparateurs des données

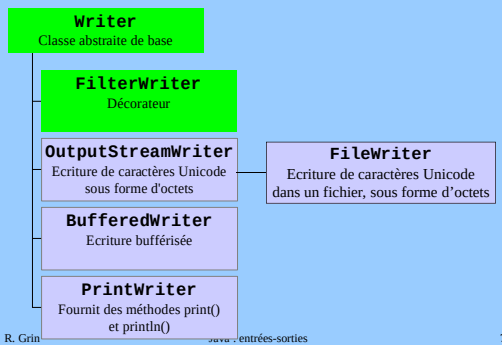
- Pour les flots d'octets, il suffit de relire les données dans l'ordre dans lequel elles ont été écrites
- Pour les flots de caractères, on doit mettre des séparateurs entre les données ; par exemple, pour distinguer un nom d'un prénom

Relire des données avec séparateurs

- Le plus simple est d'utiliser la méthode **`String[] split(String exprReg)`** de la classe **`String`** pour décomposer les lignes du flot

Écriture d'un flot de caractères

Hiérarchie des principales classes d'écriture d'un flot de caractères



Écriture dans un flot composé de lignes de texte

- La classe **`PrintWriter`** comprend les méthodes **`print`** et **`println`**

Lecture et écriture de caractères dans un fichier – codage des caractères

Codage

- En Java les caractères sont codés en Unicode
- Souvent pas le cas sur les périphériques
- Un codage par défaut est automatiquement installé par le JDK, conformément à la locale
- Il ne faut pas s'appuyer sur le codage par défaut ; **il faut indiquer explicitement le codage utilisé**

Lecture dans un fichier de texte

- **`FileReader`** lit des caractères dans un fichier, suivant le codage par défaut

Lire les lignes de texte d'un fichier

```
BufferedReader br =
  new BufferedReader(
        new FileReader("fichier"));
try {
  String ligne;
  while ((ligne = br.readLine()) != null) {
    // Traitement de la ligne
    . . .
  }
}
finally {
  if (br != null) br.close();
}
```

N'utilise pas **`EOFException`** pour repérer la fin du fichier

Lire les lignes de texte – try avec ressource du JDK 7

```
try (
  BufferedReader br =
    new BufferedReader(
        new FileReader("fichier"));
) {
  String ligne;
  while ((ligne = br.readLine()) != null) {
    // Traitement de la ligne
    . . .
  }
}
```


Écriture dans un fichier de texte

- **`FileWriter`** écrit des caractères Unicode dans un fichier, suivant le codage par défaut
- Souvent plus simple d'utiliser **`PrintWriter`** dont les méthodes ne lancent pas d'exceptions

Écrire une ligne de texte

```
PrintWriter pw = new PrintWriter("fichier");
String ligne;
. . .
pw.println(ligne);
pw.close();
```

Lecture-écriture avec un codage particulier

- Pour un autre codage que le codage par défaut, utiliser ces classes qui permettent de préciser un codage particulier :
  - en lecture **`InputStreamReader`**
  - en écriture **`OutputStreamWriter`** ou **`PrintWriter`**

Exemple

```
FileInputStream fis =
  new FileInputStream("fichier");
Reader reader =
  new InputStreamReader(
          fis,
          Charset.forName("UTF-8"));
//     ou StandardCharsets.UTF_8
```


Classe **`java.net.URL`** (*Uniform Resource Locator*)

Lire le code HTML d'une page Web

```
URL url = new URL(
      "http://www.unice.fr/index.html");
InputStream is = url.openStream();
BufferedReader br =
  new BufferedReader(
    new InputStreamReader(is));
while ((ligne = br.readLine()) != null) {
  System.out.println(ligne);
}
```


Passer de **`URL`** à **`Path`**

- Pour utiliser les méthodes de lecture ou écriture de fichiers :

```
Paths.get(url.toURI())
```


Noms de fichiers, ressources

Un problème fréquent

- Un projet fonctionne dans l'environnement de développement mais ne fonctionne plus quand l'application est distribuée sous la forme d'un fichier jar
- Les fichiers utilisés par l'application (images en particulier) ne sont plus trouvés par l'application
- La solution est d'utiliser des noms de ressource et pas des noms de fichier pour désigner ces fichiers

Le problème avec les noms de fichiers

- Si on utilise un nom absolu pour un fichier, il faut modifier et recompiler l'application dès que le fichier change de place
- Les noms relatifs posent aussi des problèmes (quel est le répertoire courant ?)
- Le fichier n'est plus trouvé s'il est mis dans un fichier jar avec le code de l'application

Une meilleure solution

- Utiliser un nom de ressource pour désigner le fichier et utiliser les méthodes de la classe **`Class`**
  `URL getResource(String `*`nom`*`)`
  ou
  `InputStream`
  `getResourceAsStream(String `*`nom`*`)`
- Le *nom de ressource* passé en paramètre indique l'endroit où la ressource sera recherchée

Nom d'une ressource

- Cet endroit dépend de la façon dont la classe a été chargée en mémoire
- Le plus souvent,
  - si le nom est absolu (commence par « / »), il est *relatif* au *classpath* (racine du jar si l'application est distribuée dans un jar)
  - si le nom est relatif, il est relatif au répertoire où se trouve le fichier .class qui contient le code

Avantage de cette solution

- Si on déplace le répertoire de l'application, les fichiers de ressources suivent et il n'est pas besoin de modifier le code
- L'application fonctionne si elle est distribuée avec ses ressources dans un fichier jar

Lire le contenu d'un fichier ressource

- Le plus simple est d'utiliser la méthode **`getResourceAsStream`** (renvoie **`null`** si la ressource n'a pas été trouvée) :

```
InputStream in = this.getClass()
  .getResourceAsStream("/rep/fich");
```

Où le fichier **`fich`** va-t-il être recherché ?

`fich` recherché dans un sous répertoire **`rep`** du *classpath*

Cas d'une méthode `static`

- En ce cas, il faut remplacer « **`this.getClass()`** » par la classe qui contient le code ; appelons-la « **`p.C1`** ») :
  `p.C1.class.getResourceAsStream(…)`

Écrire dans une ressource

- Récupérer le nom du fichier à partir de l'URL
- Exemple pour un fichier au format texte :

```
URL url =
  getClass().getResource("/fichier");
PrintWriter pw = new PrintWriter(
  Paths.get(url.toURI()).toString());
```


Sérialisation

Définition

- Sérialiser un objet c'est transformer l'état (valeurs des variables d'instance) de l'objet en une suite d'octets
- Permet de conserver l'état de l'objet pour le retrouver ensuite et reconstruire un autre objet avec le même état
- Le code de la classe n'est pas sérialisé !

`ObjectOutputStream`

```
HashMap<String,Article> map = new HashMap<>();
// remplit la table de hachage avec des objets
. . .
try (ObjectOutputStream oos =
       new ObjectOutputStream(
         new FileOutputStream("fichier.ser")))
{
  oos.writeObject(map);
  oos.writeInt(125); // on peut écrire type primitif
}
```

Sérialise la map, et tous les objets qu'elle contient

`ObjectInputStream`

```
try (ObjectInputStream ois =
       new ObjectInputStream(
         new FileInputStream("fichier.ser")))
{
  HashMap<String,Article> map =
    (HashMap<String,Article>)ois.readObject();
  int i = ois.readInt();
}
```

Renvoie un **`Object`**
Il faut *caster*

Récupère la map, et **tous les objets qu'elle contenait**

Interface `Serializable`

- Pour pouvoir être sérialisé, un objet doit être une instance d'une classe qui implémente l'interface **`Serializable`**
- Cette interface ne comporte aucune méthode ; elle sert seulement à marquer les classes sérialisables
- Le plus souvent rendre une classe sérialisable ne nécessite l'écriture d'aucune ligne de code

Mises en forme

Paquetage `java.text`

- Contient des classes pour mettre en forme les nombres et dates
- **`NumberFormat`** pour les nombres, les valeurs monétaires et les pourcentages
- **`DateFormat`** pour les dates
- Pour afficher ou enregistrer dans un fichier avec un certain format, le plus simple est d'utiliser la méthode **`printf`**

Méthodes `printf`

- Syntaxe semblable au langage C
- Dans les classes **`PrintStream`** et **`PrintWriter`**
- Types des paramètres :
  `Locale` ***`l`***`, String ` ***`format`***`, Object... ` ***`args`***
- Une variante prend la locale par défaut :
  `String ` ***`format`***`, Object... ` ***`args`***

Exemple simple

```
System.err.printf(
   "Impossible ouvrir le fichier '%s': %s",
   fileName,
   exception.getMessage());
```


Isoler les entrées-sorties

Pourquoi ?

- Les entrées-sorties sont souvent complexes
- Pas rare de changer la façon d'accéder à des données externes (fichiers, BD relationnelle, XML,…)
- Elles compliquent/ralentissent les tests
- Il vaut donc mieux isoler les entrées-sorties du reste de l'application

Expressions régulières

Généralités

- Les expressions régulières servent à
  - rechercher des chaînes de caractères correspondant à un *modèle*
  - en extraire des sous-chaînes
  - en modifier une partie

Exemples (1)

- **`a*b`** : b, ab, aaaab, ~~axyzb~~
- **`^a*b`** : idem, seulement en début de ligne
- **`[a-z&&[^bc]]`** : 1 lettre, sauf b et c
- **`.`** : n'importe quoi, sauf, éventuellement, fin de ligne
- **`<table>.*</table>`** : <table>…</table>

`<table width="750" align="center" border="1">` ?

Exemples (2)

`<table width="750" align="center" border="1">`

- **`<table.*?>(.*?)</table\\s*?>`** :
  permet des attributs dans <table>
  `\s` : caractère « blanc »
  `.*?` : n'importe quoi en mode « non glouton »
  `(.*?)` : 1er groupe de l'expression (**`\1`**)

Glouton ou pas ?

- Par défaut, la recherche englobe le plus de caractères possible, tant qu'une chaîne correspond à l'expression régulière
- Si on veut que le moins de caractères possible soient englobés, il faut ajouter **`?`** derrière un quantifieur

Exemple

- Si on cherche dans **Abcdec**,
- **"A.*c"** correspond à **Abcdec**
- **"A.*?c"** correspond à **Abc**

Classe **String**

- Contient des méthodes qui utilisent des expressions régulières
  - **boolean matches(String *expreg*)**
  - **String replaceAll(String *expreg*, String *remplacement*)**
  - Variante : **replaceFirst**
  - **String[] split(String *separateur*)**

Expression régulière

Exemple

```
String phrase =
  "Bonjour bonhomme veux-tu un bon bonbon ?";
String phraseModifiee =
  phrase.replaceAll("\\bbon(\\w+)", "mal$1");
System.out.println(phraseModifiee);
```

- affichera
  Bonjour malhomme veux-tu un bon malbon ?
- Si on veut ne pas tenir compte de la casse ou faire des traitements plus complexes, utiliser les classes **Pattern** et **Matcher** (de **java.util.regexp**)

Classe **Pattern**

- Correspond à une expression régulière « compilée », préparée pour des futures recherches accélérées
- Si une expression régulière est utilisée plusieurs fois, on a intérêt à la compiler

Méthode **compile**

- **static Pattern compile(String exprReg)** compile une expression régulière
- Si on veut ne pas tenir compte de la casse et considérer que « . » peut correspondre à une fin de ligne :

```
Pattern patternTR =
  Pattern.compile("truc(.*?)machin",
    Pattern.CASE_INSENSITIVE |
    Pattern.DOTALL);
```


Obtenir un **Matcher**

- **Matcher matcher(CharSequence *texte*)** renvoie un **Matcher** pour rechercher l'expression régulière dans ***texte***
- Les chaînes de caractères dans lesquelles on recherche une expression régulière sont du type **CharSequence**

Méthodes de la classe `Matcher`

- **`lookingAt`**, **`find`** et **`matches`** renvoient **`true`** si une expression régulière a été trouvée
- **`replaceAll`** , **`replaceFirst`**, **`appendReplacement`** et **`appendTail`** remplacent l'expression régulière par une chaîne de caractères

Groupes

- Une expression régulière peut contenir des portions entre parenthèses
- **`\bbon(\w*)`** correspond aux sous-chaînes formées d'un mot qui commence par « bon » ; **`\w*`** désigne le reste du mot
- Les portions entre parenthèses sont appelées des groupes et sont numérotées par ordre d'apparition de leur parenthèse ouvrante

Méthodes pour les groupes

- **`String group(int n)`** renvoie la sous-chaîne qui correspond au **n**ième groupe
- On obtient le numéro du caractère qui débute et termine (+ 1) chaque groupe avec les méthodes **`int start(int n)`** et **`int end(int n)`**

Exemple de code

```
String phrase =
  "Bonjour bonhomme veux-tu un bon bonbon ?";
String expReg = "\\bbon(\\w+)";
Pattern p = Pattern.compile(
  expReg, Pattern.CASE_INSENSITIVE);
Matcher m = p.matcher(phrase);
System.out.println(m.replaceAll("mal$1"));
System.out.println(m.matches()); // false
System.out.println(m.lookingAt()); // true
System.out.println(m.find()); // true
m.reset(); // repart au début
```


Fin du code

```
// Affiche détails de la recherche
while (m.find()) {
  System.out.print("Trouvé " + m.group()
                   + " en position "
                   + m.start());
  System.out.println(" suffixe : "
                     + m.group(1)
                     + " en position "
                     + m.start(1));
}
```


Affichage de l'exemple

```
maljour malhomme veux-tu un bon malbon ?
false
true
true
Trouvé Bonjour en position 0 ; suffixe :
  jour
Trouvé bonhomme en position 8 ; suffixe :
  homme
Trouvé bonbon en position 32 ; suffixe :
  bon
```

Remplacement

- La méthode **`appendReplacement`** offre plus de souplesse que **`replaceAll`** : elle ajoute dans un **`StringBuffer`**, en effectuant les remplacements un à un
- On peut choisir ce que l'on fait entre chaque remplacement

Classe `File`

Classe `File`

- Représente un chemin de fichier, indépendamment du système d'exploitation
- Remplacé par **`Path`** dans le JDK 7