Entrées-sorties

Université Française d'Egypte
Version O 7.5 – 5/10/15
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
- Expressions régulières

- Un programme utilise le plus souvent des données externes venant
  - du Web
  - de bases de données
  - de simples fichiers
  - d'appareils de mesure
  - …
- Ce support étudie comment utiliser les fichiers, et un peu le Web

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

Paquetages

- API contenue dans le paquetage **`java.nio.file`**
- Complète et améliore **`java.io`**

Principales 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
- Lire et écrire le contenu de fichiers

Interface de base : **`Path`**

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

Création d'un **`Path`**

Sous Windows, **`/rep/truc`** ou **`\\rep\\truc`**

- **`Path path = Paths.get("/rep/truc");`**
- **`Path path = Paths.get("/rep", "truc");`**
- Chemin relatif par rapport à un répertoire dont on a déjà un **`Path`** :

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

- Comme **`resolve`** mais part du répertoire parent de **`path2`** :

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


Exemples

```
Path path1 = Paths.get("/rep/truc");
Path path2 = Paths.get("/rep", "truc");
System.out.println(path1 + "; " + path2);
System.out.println(path1.equals(path2));
Path rep = Paths.get("/rep");
Path path3 = rep.resolve("truc");
System.out.println(path2.equals(path3));
Path path4 = path1.resolveSibling("machin");
System.out.println(path4);
System.out.println(
  Paths.get("/rep/truc").resolve("machin")
       .resolveSiblings("bidule");
```

`\rep\truc; \rep\truc`

`true`

`true`

`\rep\machin`

On peut chaîner les appels

`\rep\truc\bidule`

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\fich`**, **`getRoot()`** retourne **`C:\`**

Conditions sur le nom de fichier

- **`boolean startsWith`** (et **`endsWith`**) ; paramètre **`String`** ou **`Path`** ; attention, si le paramètre est de type **`String`**, il est d'abord transformé en **`Path`**
- Exemples : **`true`** ou **`false`** ?

```
Paths.get("/rep/fichier.txt")
     .endsWith(".txt")
```

`false` !

```
Paths.get("/rep/fichier.txt")
   .endsWith("fichier.txt")
```

`true`

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 de Java

- **`user.dir`** : répertoire courant

```
String rep =
    System.getProperty("user.dir");
```

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

Dans les IDE

- Dans une application lancée sous NetBeans ou Eclipse
  - le répertoire courant est le répertoire qui contient le projet (le répertoire père de **`src`**)
  - le *classpath* est le répertoire **`src`**

A noter

- Souvent intéressant de considérer les fichiers comme des ressources désignées par un **`URL`** plutôt que par un **`Path`**
- On verra dans la section « Ressources » comment passer de **`URL`** à **`Path`** pour utiliser les méthodes de **`Files`**

Classe **`Files`**

- Fournit des méthodes **`static`** pour
  - lire, écrire
  - manipuler (copier, déplacer, supprimer)

  des fichiers ordinaires et des répertoires
- La plupart des méthodes de **`Files`** peuvent lancer une **`IOException`** ; elles tiennent compte de l'existence des fichiers
- Exemple :
  `Files.delete(`*`path`*`);`

Vérifications sur les fichiers

- Existence :
  `exists(Path)`, **`notExists(Path)`**
- Autorisations :
  `isReadable(Path)`, **`isWritable(Path)`**, **`isExecutable(Path)`**

Gestion des fichiers

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

Exemple

```
Files.move(source,
  autreRep.resolve(source.getFileName()),
  StandardCopyOption.REPLACE_EXISTING);
```

source et **autreRep** sont de type **Path**

Qu'est-ce que ça fait ?

Glob

- Pour filtrer les noms de fichiers
- Exemples :
  - ***** : 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

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

Interface implémentée par **DirectoryStream** ?

Que fait ce code ?

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

Lire et écrire des « petits » fichiers

- **Files** contient des méthodes pour
  - lire ou écrire tout le contenu d'un fichier *en une fois*
  - en prenant en charge l'*ouverture et la fermeture des fichiers*
- Très pratiques mais ne peuvent pas être utilisées pour les très gros fichiers, car tout le contenu du fichier est enregistré dans la mémoire centrale

2 types de fichiers

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

Lire des petits fichiers

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

Exemple de lecture d'un fichier texte

```
Path = Paths.get("unFichier.txt");
List<String> lignes = Files.readAllLines(
    path, StandardCharsets.UTF_8);
for (String ligne : lignes) {
  // traiter une ligne lue
  ...
}
```

Ne pas oublier de traiter les **`IOException`** (laisser remonter ou attraper)

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

UTF 8 par défaut

Interface **`CharSequence`**

- Interface utilisée pour les signatures des méthodes qui travaillent sur des suites de **`char`**
- Implémentée par **`String`** et **`StringBuilder`**

Options pour écrire

- Par défaut un fichier existant est écrasé
- Pour ajouter à la fin du fichier :
  `Files.write(path, bytes, StandardOpenOption.APPEND);`
- **`java.nio.file.StandardOpenOption`** est une énumération qui contient des valeurs pour les options (par défaut **`CREATE`**, **`TRUNCATE_EXISTING`** et **`WRITE`**)

Exemple d'écriture dans un fichier texte

```
List<String> lignes = new ArrayList<>();
// Remplit la liste des lignes
...
Files.write(
  Paths.get(cheminFichier),
  lignes,
  Charset.defaultCharset());
```

Pour les plus gros fichiers

- Les méthodes qui viennent d'être étudiées sont à privilégier car le code est simple
- Si le fichier est trop gros pour tenir en mémoire centrale ou si on veut plus de souplesse il faut utiliser d'autres 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

Streams de Java 8

UTF 8 par défaut

- Depuis Java 8 (avec une variante qui prend une **`String`** en paramètre) dans **`Files`** : **`Stream<String> lines(Path, Charset)`**
- **`BufferedReader`** a aussi une méthode **`lines()`** qui retourne un **`Stream<String>`** ; ne pas toucher au reader pendant l'utilisation du stream
- Peuvent lancer **`IOException`**

Exemple avec Stream et lambda

Que fait ce code ?

```
String laPlusLongue =
  Files.lines(path)
  .max(
    Comparator.comparing(l -> l.length()))
  .get();
```

Pourquoi ce **`get`** ?

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
- Remarque : ici « **`Stream`** » n'a pas le même sens que les streams étudiés avec les expressions lambda ; il s'agit de flots d'octets

Flots (*streams*)

Présentation

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

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 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 (méthode **`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
  - …

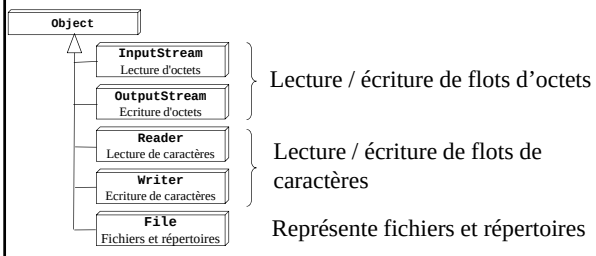
Exemple de décoration

```
Path path = Paths.get("fichier");
try (
  BufferedInputStream bis =
    new BufferedInputStream(
      Files.newInputStream(path))) {
  int octet;
  while ((octet = bis.read()) != -1) {
     ...
  }
}
```

Fin du fichier

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

Classes de base de **`java.io`**



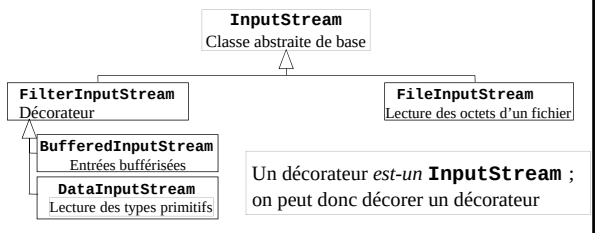
Lecture / écriture de flots d'octets

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



Un décorateur *est-un* **`InputStream`** ; on peut donc décorer un décorateur

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 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 *est-un* **`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`**

Ferme tous les flots

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

`Files` et décorateurs

- Évidemment on peut décorer les classes récupérées par les méthodes de **`Files`** telles **`newBufferedReader`** ou **`newOutputStream`**

Écriture d'un flot d'octets

Écrire des types primitifs dans un fichier

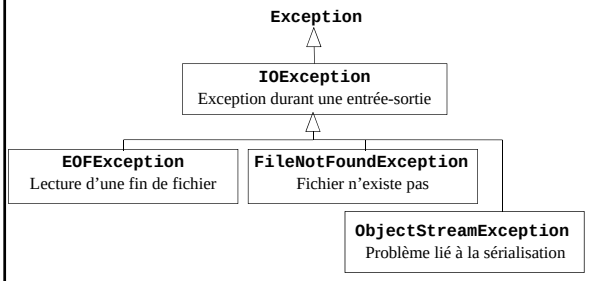
```
DataOutputStream dos =
  new DataOutputStream(
    new BufferedOutputStream(
      new FileOutputStream("fichier")));
dos.writeDouble(12.5);
dos.writeUTF("Dupond");
dos.writeInt(1254);
dos.close();
```


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

```
public void lire(String fich) 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 et écriture de caractères dans un fichier – codage des caractères

Lecture d'un flot composé de lignes de texte

- La classe **`BufferedReader`** contient 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

- **`String[] split(String exprReg)`** de la classe **`String`** permet de décomposer les lignes lues dans le flot

Écriture dans un flot composé de lignes de texte

- Le plus simple est d'écrire avec les méthodes **`print`** et **`println`** de la classe **`PrintWriter`**

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

Ancienne version

```
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 d'un fichier

Nouvelle version

```
try (
  BufferedReader br =
    Files.newBufferedReader(
      Paths.get("fichier"))) {
  String ligne;
  while ((ligne = br.readLine()) != null) {
    // Traitement de la ligne
    . . .
  }
}
```

Préférable d'indiquer explicitement le codage en paramètre de **`newBufferedReader`**

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 des lignes de texte

```
try (
  PrintWriter pw = new PrintWriter("fichier")
) {
  String ligne;
  . . .
  pw.println(ligne);
}
```

Pour ne pas écraser un ancien fichier utiliser le constructeur **`PrintWriter(OutputStream)`** ou **`Files.newOutputStream`**

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 l'utiliser ; **il faut indiquer explicitement le codage utilisé**

Utiliser un codage particulier

- Le plus simple est d'utiliser les méthodes de **`Files`**, par exemple **`newBufferedReader(Path, Charset)`**
- Les transparents suivants montrent comment faire avec du code écrit sans utiliser la classe **`Files`**

Lecture-écriture avec un codage particulier

- Ces classes permettent de préciser un codage particulier (si on n'a pas utilisé **`Files`**) :
  - 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 de **`Files`** :

```
Path path = 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 si ces fichiers sont dans le jar

La solution

- Utiliser un nom de ressource pour désigner le fichier avec 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 ; marche si la ressource est dans un jar ou non

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

Lire le contenu d'un fichier ressource

- Le plus simple est d'utiliser la méthode **`getResourceAsStream`** (retourne **`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`**

- Remplacer « **`this.getClass()`** » par la classe qui contient le code ; appelons-la « **`p.C1`** ») :

```
p.C1.class.getResourceAsStream(…)
```


Écrire dans une ressource

- Exemple pour un fichier au format texte :

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


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
}
```

Décorateur qui sait sérialiser

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, 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());
```


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]`** : 1 lettre minuscule
- **`.`** : n'importe quoi
- **`<table>.*</table>`** : <table>…</table>

Ne retrouve pas
`<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
- Par 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 séparateur)`**

Expression régulière

Exemple

`\b` : début de mot
`\w` : lettre, chiffre ou « _ »

```
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 **`Pattern.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`** (début du texte) et **`matches`** (tout le texte) retournent **`true`** si l'expression régulière a été trouvée
- **`find`** se cale sur la prochaine occurrence de l'expression régulière (utilisé le plus souvent dans une boucle) ; retourne **`false`** s'il ne reste plus d'occurence
- **`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)`** retourne 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)`**
- **`String group()`** retourne la chaîne du texte qui correspond à l'expression régulière

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);
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 ?

```
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