Entrées-sorties 1

Université de Nice - Sophia Antipolis
Version 3.1.1 – 29/1/12
Richard Grin

Plan de cette partie 1

- Gestion des fichiers (remplace la classe File)
- Les flots (*streams*), modèle de conception « décorateur »
- Classe URL
- Noms de fichiers, ressources
- Sérialisation
- Analyse lexicale
- Clavier – écran
- Classe File

Gestion des fichiers Classes `Path` et `Files`

Introduction

- Nouvelle API **NIO.2** introduite par le JDK 7, contenue dans le paquetage **`java.nio.file`**
- Elle remplace en particulier la classe **`java.io.File`** (qui est donc maintenant à éviter) qui est présentée à la fin de ce support

Fonctionnalités (1/2)

- Manipulation des noms de fichier
- Opérations de base sur les fichiers : copier, déplacer, supprimer
- Lister les fichiers d'un répertoire, créer et supprimer un répertoire
- Afficher et modifier les métadonnées sur les fichiers (autorisations, propriétaire,…)

Fonctionnalités (2/2)

- Lire et écrire le contenu de petits fichiers
- Traverser une arborescence de fichiers
- Surveiller les changements dans un répertoire

Interface de base : **`Path`**

- Correspond à un nom/chemin de fichier relatif ou absolu dans un système de fichiers
- **Indépendant de l'existence ou non d'un fichier qui a ce nom**
- Peut représenter un lien symbolique
- Dans la suite, « instance de **`Path`** » signifiera « instance d'une classe qui implémente **`Path`** »

Chaînage des méthodes

- Plusieurs méthodes de **`Path`** renvoient un **`Path`**, ce qui permet de chaîner les appels de méthode

Classe **`Paths`** - création d'un **`Path`**

- La classe **`Paths`** contient 2 méthodes **`get static`** pour créer une instance de **`Path`** à partir d'une ou plusieurs **`String`** ou d'un **`URI`** (voir « URL et URI » plus loin sur ce support)
- **`Path get(String, String...)`** :
  - s'il n'y a qu'un paramètre, il décrit le chemin
  - sinon le chemin est constitué des différents paramètres

Classe **`Paths`** - création d'un **`Path`**

- **`Path.get`** est un raccourci pour **`FileSystems.getDefault().getPath`**
- Si on a déjà un **`Path`** pour le répertoire du **`Path`** à créer, il est plus souple d'utiliser la méthode **`resolve`**

Exemples

- **`Path path = Paths.get("/rep/truc");`**
- **`Path path = Paths.get("/rep", "truc");`** correspond au même chemin que l'exemple précédent
- Sous Windows, le paramètre de type **`String`** peut être donné avec le séparateur « / » ou « \\ » (il faut doubler le « \ » mais les transformations de chemin sont données/affichées avec le séparateur du système utilisé)

Joindre 2 chemins

- **`path.resolve(pathRelatif)`** retourne un nouveau **`Path`** en prenant pour base **`path`** et en y ajoutant **`pathRelatif`**
- **`path.resolveSibling(pathRelatif)`** retourne un nouveau **`Path`** en prenant pour base le répertoire parent de **`path`**

Itinéraire entre 2 chemins

- **`Path relativize(Path)`** retourne le chemin relatif pour aller de **`this`** au paramètre (inverse de resolve)
- Exemple : si **`path`** correspond à /a/b et si le paramètre correspond à /a/b/c/d, **`relativize`** renvoie le chemin qui correspond à c/d
- Le chemin renvoyé peut comporter des « .. »

Comparaison de chemins

- **`boolean equals(Object)`** (redéfinit le **`equals`** de **`Object`**)
- **`boolean startsWith`** ; paramètre **`String`** ou **`Path`**
- **`boolean endsWith`** ; paramètre **`String`** ou **`Path`**
- **`int compareTo(Path)`** (de l'interface **`Comparable<Path>`**)

Informations sur un **`Path`**

- **`Path getFileName()`** : nom terminal du fichier
- **`Path getName(i)`** : i^{ème} élément du chemin
- **`int getNameCount()`** : nombre d'éléments
- **`Path subPath(début, fin)`** : éléments compris entre début (compris) et fin (pas compris), sans la racine
- **`Path getRoot()`** : racine du chemin
- **`boolean isAbsolute()`** : indique si le chemin est absolu

Exemples

	/home/dupond/fich	**dupond/fich**
toString()	/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 Window C:\home\dupond\fichier, getRoot renvoie C:\

Normalisation d'un **`Path`**

- Normalisation : enlever les « . » et « .. » qui ne servent à rien
- Exemple :
  `Paths.get("/home/dupond/../fich")`
  `.normalize()`
  correspond à /home/fich

Conversion en chemin absolu

- Utilise le répertoire courant
- **`Path toAbsolutePath()`**
  Le fichier peut exister ou ne pas exister
- **`Path toRealPath(LinkOption.. options)`**
  Lance **`NoSuchFileException`** si le fichier n'existe pas ; donne le même résultat que **`toAbsolutePath()`** sinon
  Avec des options pour suivre ou non les liens symboliques (dépendant du système d'exploitation)

Conversion d'un **`Path`**

- En **`URI`** (correspond à une adresse pour un navigateur) : **`URI toUri()`**
  Type de résultat :
  file:///C:/home/dupond/fich
  Utilise le répertoire courant si le chemin est relatif
  Ne tient pas compte de l'existence ou non du fichier correspondant au chemin
- En **`File`** (pour interopérabilité avec ancien code) : **`File toFile()`**

Rappel : quelques propriétés système

- **`user.dir`** : répertoire courant
- **`file.separator`** : caractère pour séparer les différentes parties d'un nom de fichier (/ pour Unix, \ pour Windows) ; aussi fourni par **`FileSystems.getDefault().getSeparator()`**
- **`user.home`** : répertoire « home » de l'utilisateur
- Exemple : **`System.getProperty("user.dir")`**

Parenthèse sur les IDE

- Le répertoire courant quand on lance une application sous Eclipse est le répertoire qui contient le projet (le répertoire père de **`src`**)
- Le classpath est le répertoire **`src`** (en fait le répertoire **`bin`** pendant l'exécution)
- Situation similaire pour NetBeans
- Attention, vérifiez que votre application fonctionne toujours après l'avoir mise dans un jar, lorsque vous la lancez en dehors d'un IDE

Opérations sur les fichiers

- La classe **`Files`** (avec un s final) fournit des méthodes **`static`** pour lire, écrire et manipuler des fichiers ordinaires et des répertoires
- La plupart des méthodes de **`Files`** peuvent lancer une **`IOException`**, ou plus spécifiquement une **`java.nio.file.FileSystemException`**
- Tiennent compte de l'existence des fichiers

`OpenOption`

- Plusieurs méthodes de **`Files`** ont un paramètre de type **`OpenOption`**
- L'énumération **`StandardOpenOption`** implémente l'interface **`OpenOption`** ; elle définit les valeurs **`WRITE`**, **`APPEND`**, **`TRUNCATE_EXISTING`**, **`CREATE_NEW`**, **`CREATE`**, **`DELETE_ON_CLOSE`**, **`SPARSE`**, **`SYNC`**, **`DSYNC`**

Vérifications sur les fichiers

- Vérifier l'existence (liens symboliques suivis ou non) : **`exists(Path, LinkOption)`** et **`notExists(Path, LinkOption)`**
- Vérifier les autorisations (voir aussi la suite sur les métadonnées) : **`isReadable(Path)`**, **`isWritable(Path)`**, **`isExecutable(Path)`**
- Vérifier si 2 chemins représentent le même fichier : **`isSameFile(Path, Path)`**

Supprimer un fichier ou un répertoire

- **`void delete(Path)`** : lance **`NoSuchFileException`** si le fichier n'existe pas
- **`boolean deleteIfExist(Path)`** : idem mais ne lance pas une exception si le fichier n'existe pas (renvoie **`true`** si le fichier a été supprimé)

Copier un fichier ou un répertoire

- **`Path copy(Path, Path, CopyOption…)`** retourne le chemin de la cible (pour chaînage)
- Les options possibles :
  - **`REPLACE_EXISTING`**
  - **`COPY_ATTRIBUTES`**
  - **`NOFOLLOW_LINKS`**
- On peut aussi faire des copies entre un flot et un fichier (dans les 2 sens : avec **`InputStream`** et **`OutputStream`**)

Déplacer un fichier ou un répertoire

- **`Path move(Path, Path, CopyOption…)`**
- Les options possibles :
  - **`REPLACE_EXISTING`** : écrase un éventuel fichier existant
  - **`ATOMIC_MOVE`** : l'opération est complètement exécutée, ou pas du tout

Créer un fichier ou un répertoire

- **`Path createFile(Path, FileAttribute<?>…)`**
- **`Path createDirectory(Path, FileAttribute<?>…)`**
- **`Path createDirectories(Path, FileAttribute<?>…)`** : crée avant tous les répertoires intermédiaires s'ils n'existent pas (idem **`mkdir -p`** d'Unix)

Exemple

```
Set<PosixFilePermission> perms =
  PosixFilePermissions
    .fromString("rwxr-x---");
FileAttribute<Set<PosixFilePermission>> attr =
  PosixFilePermissions.asFileAttribute(perms);
Files.createDirectory(file, attr);
```


Créer un lien

- **`Path createLink(Path `*`lien`*`, Path `*`existant`*`)`** : crée un lien « hard » (le 1er paramètre) du 2ème paramètre
- **`Path createSymbolicLink(Path `*`lien`*`, Path `*`fichierPointé`*`, FileAttribute<?>…)`**

Créer un fichier temporaire

- **`Path createTempFile(Path rep, String préfixe, String suffixe, FileAttribute<?>)`** : crée un fichier temporaire ; utilise « .tmp » si le suffixe est null (voir javadoc pour détails)
- **`Path createTempFile(String suffixe, FileAttribute<?>)`** : crée le fichier temporaire dans le répertoire par défaut des fichiers temporaires (/tmp ou /var/tmp sous Unix, C:\WINNT\TEMP sous Windows)

Gérer les métadonnées sur les fichiers

- La classe **`Files`** a aussi des méthodes pour obtenir ou modifier les métadonnées sur les fichiers et les répertoires

Lire les métadonnées sur les fichiers

- Les méthodes pour obtenir les métadonnées sont nombreuses et ne seront pas détaillées ici : **`size`**, **`isDirectory`**, **`isRegularFile`**, **`isSymbolicLink`**, **`isHidden`**, **`getOwner`**, **`getLastModifiedTime`**, **`getPosixFilepermissions`**, **`getAttribute`**, **…**
- Il est possible de récupérer des groupes d'attributs en une seule fois avec les méthodes **`readAttributes`**

Exemple

```
Path fichier = ...;
BasicFileAttributes attr =
    Files.readAttributes(
      fichier,
      BasicFileAttributes.class);
System.out.println("creationTime: " +
                   attr.creationTime());
System.out.println("lastAccessTime: " +
                  attr.lastAccessTime());
```


Modifier des métadonnées

- Certaines métadonnées peuvent être modifiées avec les méthodes suivantes : **`setLastModifiedTime`**, **`setOwner`**, **`setAttribute`**
- Consultez le tutoriel en ligne d'Oracle pour plus de détails : http://docs.oracle.com/javase/tutorial/essential/io/fileAttr.html

Exemple 1

```
Path sourceFile = ...;
Path newFile = ...;
PosixFileAttributes attrs =
  Files.readAttributes(
    sourceFile,
    PosixFileAttributes.class);
FileAttribute<Set<PosixFilePermission>> attr =
  PosixFilePermissions
    .asFileAttribute(attrs.permissions());
Files.createFile(file, attr);
```

Exemple 2

```
Path file = ...;
Set<PosixFilePermission> perms =
  PosixFilePermissions
    .fromString("rw-------");
FileAttribute<Set<PosixFilePermission>> attr =
  PosixFilePermissions
    .asFileAttribute(perms);
Files.setPosixFilePermissions(file, perms);
```


Glob

- Modèle qui ressemble à une expression régulière, mais pour filtrer les noms de fichiers (attention, la signification des caractères spéciaux est différente de celle des expressions régulières)
- 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

Exemples de Glob

- **`[a-g,A-G]`** : on peut donner plusieurs segments
- **`{abc, ABC, xyz}`** ou **`{temp*, tmp*}`** : collection de sous-modèles
- **`\`** : pour enlever la signification spéciale du caractère suivant (**`\[`** ou **`\\`** par exemple)
- Détails de la syntaxe dans la javadoc de la méthode **`getPathMatcher`** de la classe **`java.nio.file.FileSystem`**

Interface **`PathMatcher`**

- **`PathMatcher`** compare une **`String`** à un glob ou à une expression régulière (voir partie 2 de ce support de cours) :
  ```
  FileSystems.getDefault()
    .getPathMatcher("glob:" + modele);
  ```
  (on peut remplacer **`glob`** par **`regex`**)
- Cette interface contient la méthode **`boolean matches(Path chemin)`** qui renvoie **`true`** si le chemin correspond au modèle

Liste des fichiers d'un répertoire

- Pour obtenir des performances correctes, même pour les répertoires qui contiennent de nombreux fichiers, les fichiers d'un répertoire sont fournis sous la forme d'un flot : **`DirectoryStream<Path>`** (ne pas oublier de fermer le flot après usage)
- Cette classe implémente **`Iterable<Path>`**, ce qui simplifie son utilisation (utilisation d'une boucle *for-each*)

Méthodes pour lister un répertoire

- 3 méthodes de **`Files`** fournissent un tel flot, suivant que l'on veut avoir tous les fichiers ou seulement des fichiers sélectionnés ; ces méthodes peuvent lancer une **`IOException`**
- La classe interne **`DirectoryStream.Filter`** permet de sélectionner les fichiers sur un critère quelconque ; il suffit d'implémenter la méthode **`boolean accept(Path fichier)`** pour qu'elle renvoie **`true`** pour les fichiers sélectionnés

Méthodes pour lister un répertoire

- **`newDirectoryStream(Path rep)`** : avoir tous les fichiers du répertoire
- **`newDirectoryStream(Path rep, String glob)`** : tous les fichiers dont le nom correspond au glob
- **`newDirectoryStream(Path rep, DirectoryStream.Filter<? super Path> filter)`** : tous les fichiers sélectionnés par le filtre

Exemple d'utilisation du flot

```
Path rep = ...;
try (DirectoryStream<Path> flot =
       Files.newDirectoryStream(rep)) {
  for (Path fich: flot ) {
    System.out.println(fich.getFileName());
  }
} catch (IOException |
         DirectoryIteratorException x) {
  // IOException ne peut être lancée que
  // par newDirectoryStream
  ...
}
```


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) {
  ...
}
```


Exemple avec filtre – le filtre

```
DirectoryStream.Filter<Path> filter =
  newDirectoryStream.Filter<Path>() {
    public boolean accept(Path fich)
      throws IOException {
        try {
          return (Files.isDirectory(fich));
        } catch (IOException x) {
            ...
        }
    }
};
```


Exemple avec filtre (suite)

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

Méthodes diverses de **`Files`**

- **`String probeContentType(Path)`** tente de déterminer le type MIME d'un fichier
- **`FileStore getFileStore(Path)`** retourne le système de fichiers qui contient le fichier passé en paramètre
- La classe **`FileStore`** peut fournir des informations sur le système de fichiers (valeurs en octets) : **`getTotalSpace()`**, **`getUsableSpace()`** (évaluation non sûre) et **`getUnallocatedSpace()`**

Obtenir tous les répertoires racines

```
Iterable<Path> dirs =
  FileSystems.getDefault()
    .getRootDirectories();
for (Path name: dirs) {
  System.out.println(name);
}
```


Visiter une arborescence de fichiers

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

Démarrer la visite

- **`walkFileTree(Path, FileVisitor<? super Path>)`** : visite toute l'arborescence placée sous le 1er paramètre et ne suit pas les liens symboliques
- **`walkFileTree(Path, Set<FileVisitOption>, int, FileVisitor<? super Path>)`** : on peut indiquer par une option si on suite les liens symboliques et indiquer aussi la profondeur des sous-répertoires à visiter (0 = on ne visite que le fichier indiqué par le 1er paramètre)

Interface **`FileVisitor<T>`**

- **`preVisitDirectory(T rep, BasicFileAttributes attrs)`** : appelée juste avant la visite des fichiers d'un répertoire
- **`visitFile(T fichier, BasicFileAttributes attrs)`** : appelée pour tous les fichiers rencontrés
- **`visitFileFailed (T rep, IOException ex)`** : appelée lorsqu'un fichier (ordinaire ou répertoire) ne peut être visité à cause de l'exception **`ex`**
- **`postVisitDirectory (T rep, IOException ex)`** : appelée juste après la visite des fichiers d'un répertoire ; **`ex`** est l'exception qui a interrompu la visite de ce répertoire (**`null`** si pas de problème)

Énumération **`FileVisitResult`**

- Toutes les méthodes de l'interface **`FileVisitor`** renvoient une valeur de cette énumération pour indiquer comment la visite doit se poursuivre
- Les valeurs :
  - **`CONTINUE`** : continuer normalement
  - **`SKIP_SIBLINGS`** : sauter les fichiers ordinaires situés dans le même répertoire
  - **`SKIP_SUBTREE`** : sauter toutes les entrées de ce répertoire (y compris les répertoires)
  - **`TERMINATE`** : fin de la visite…

SimpleFileVisitor<T>

- Classe qui implémente **FileVisitor<T>** ; il suffit au développeur de redéfinir une ou plusieurs des méthodes
- **preVisitDirectory** : retourne **CONTINUE**
- **visitFile** : retourne **CONTINUE**
- **visitFileFailed** : relance **ex**
- **postVisitDirectory** : retourne **CONTINUE** ou relance **ex**

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 ; quand un nouveau plugin est déposé, il doit être pris en compte par l'application
- L'interface **WatchService** sert à surveiller des objets (un répertoire pour cet exemple)

Surveiller un répertoire

1. Créer un surveillant :
```
WatchService watcher =
  FileSystems.getDefault()
    .newWatchService();
```
2. Enregistrer les objets à surveiller ; ils doivent implémenter l'interface **Watchable**, ce qui est le cas de **Path** qui contient 2 méthodes **register** pour s'enregistrer (nous étudierons la plus simple qui est suffisante)

Indiquer le répertoire à surveiller

- ```
  WatchKey register(
    WatchService watcher,
    WatchEvent.Kind<?>... events)
  throws IOException
  ```
- Le 2ème paramètre indique quel type d'événement le watcher va surveiller ; pour cela la classe **StandardWatchEventKinds** définit 4 constantes de type **WatchEvent.Kind<Path>** : **ENTRY_CREATE**, **ENTRY_DELETE**, **ENTRY_MODIFY**, **OVERFLOW**
  ```

Être prévenu d'une modification

- La clé renvoyée par la méthode **`register`** sert à identifier l'enregistrement
- Au départ elle est dans l'état « *ready* » ; si l'événement enregistré survient, elle passe à l'état « *signaled* » et elle est mise en file d'attente pour être retrouvée par une des méthode **`poll`** ou **`take`** de **`WatchService`**

Être prévenu d'une modification

- Il faut interroger le surveillant à intervalles réguliers avec une des méthodes suivantes :
  - **`WatchKey poll()`** : récupère une clé qui représente une modification ; retourne immédiatement la valeur **`null`** si aucune modification ; on peut passer 2 paramètres pour indiquer un timeout (valeur et unité)
  - **`WatchKey take()`** : attend une modification

Exemple schématique

```
for (;;) {
  WatchKey key = watcher.take();
  for (WatchEvent<?> evenement:
       key.pollEvents()) {
    ... // traiter l'événement
  }
  // réinitialise la clé
   boolean valid = key.reset();
   if (!valid) {
     // l'objet n'est plus enregistré
   }
}
```


Analyser un événement

- **`key.pollEvents()`** retourne une **`List<WatchEvent<?>>`**
- La classe **`WatchEvent`** contient les méthodes
  - **`kind()`** : renvoie le type d'événement
  - **`count()`** : événement répété si > 1
  - **`context()`** : dans le cas d'un **`WatchEvent<Path>`** c'est un **`Path`** qui désigne le fichier qui a provoqué l'événement

Exemple

```
// Génère un avertissement à la compilation
WatchEvent<Path> ev =
    (WatchEvent<Path>)evenement;
Path nom = ev.context();
Path fichier = dir.resolve(name);
// Traite la modification sur le fichier
...
```


Lire et écrire le contenu d'un fichier

- Plusieurs méthodes de **`Files`** facilitent (par rapport au JDK 6) la lecture et l'écriture de fichiers pour les cas les plus courants

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
- Elles sont très pratiques pour lire ou écrire les petits fichiers en une fois mais ne peuvent être utilisées pour les gros fichiers, puisque le contenu du fichier doit être enregistré dans la mémoire centrale

Lire et écrire des petits fichiers – cas où le fichier n'existe pas

- Pour la lecture, une exception est lancée si le fichier n'existe pas
- Pour l'écriture, le fichier est créé s'il n'existe pas

Lire des petits fichiers

- **`byte[] readAllBytes(Path) throws IOException`** : lit tous les octets d'un fichier
- **`List<String> readAllLines(Path, Charset) throws IOException`** : lit toutes les lignes d'un fichier texte
  Le **`Charset`** indique le codage des caractères ; **`Charset.defaultCharset()`** renvoie le codage par défaut qui dépend du système d'exploitation et de la « locale » (pays) ; voir annexe du support « Java de base »

Écrire des petits fichiers

- **`Path write(Path, byte[], OpenOption…)`** : écrit tous les octets du tableau dans un fichier
- **`Path write(Path fichier, Iterable<? extends CharSequence> `*`lignes`*`, CharSet, OpenOption…)`** : écrit toutes les lignes dans un fichier (**`CharSequence`** est une interface implémentée par **`String`** et **`StringBuilder`**)

Options pour écrire

- Les options par défaut sont **`CREATE`**, **`TRUNCATE_EXISTING`** et **`WRITE`** : un fichier existant sera écrasé
- Pour ajouter à la fin du fichier :

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


Pour les plus gros fichiers

- On peut se trouver dans un cas particulier ou il peut ne pas être possible ou intéressant d'avoir tout le contenu d'un fichier en mémoire centrale
- En ce cas, d'autres classes permettent d'écrire ou de lire les fichiers d'une façon plus souple

Pour les plus gros fichiers

- Les transparents suivants montrent comment des méthodes de la classe **`Files`** permettent d'obtenir des classes pour lire ou écrire dans un fichier d'une manière plus souple
- Les détails sur l'utilisation de ces classes seront fournis dans la section suivante (« Les flots ») de ce support de cours

Cas où le fichier n'existe pas

- Même comportement pour la lecture et l'écriture des petits fichiers :
- Pour la lecture, une exception est lancée si le fichier n'existe pas
- Pour l'écriture, le fichier est créé s'il n'existe pas

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`**

Fichiers à accès direct

- Avec le développement des bases de données relationnelles ils sont moins utilisés qu'avant mais peuvent encore être utiles
- Ils permettent un accès direct (non séquentiel), en lecture, écriture ou lecture/écriture à une partie d'un fichier

Fichiers à accès direct

- Les méthodes **`newByteChannel`** de la classe **`Files`** renvoient un **`SeekableByteChannel`** qui permet un accès direct à un fichier
- La classe **`FileChannel`** implémente cette interface ; pour le système de fichiers par défaut, il est possible de caster en **`FileChannel`** ce que renvoie **`newByteChannel`**

Interface **`SeekableByteChannel`**

- **`long position()`** : retourne la position dans le canal (un fichier pour ce cas)
- **`SeekableByteChannel position(long)`** : change la position dans le fichier
- **`int read(ByteBuffer)`** : lit des octets du fichier pour les mettre dans le buffer
- **`int write(ByteBuffer)`** : écrit dans le fichier des octets du buffer

Interface **`SeekableByteChannel`**

- **`long size()`** : retourne la taille du fichier
- **`SeekableByteChannel truncate(long)`** : tronque le fichier à la taille passée en paramètre

`FileChannel`

- Permet de lire et d'écrire n'importe où dans un fichier
- Une région du fichier peut être bloquée pour empêcher l'accès aux autres programmes
- Voir javadoc pour plus de détails
- Le transparent suivant donne un exemple de traitement d'un fichier qui contient la ligne suivante : « Bonjour monsieur »

Exemple (début)

```
Path fichier = Paths.get("…");
ByteBuffer bb1 =
  ByteBuffer.wrap("W".getBytes());
ByteBuffer bb2 = ByteBuffer.allocate(5);
try (FileChannel fc =
      (FileChannel)Files.newByteChannel(
        fichier, StandardOpenOption.READ,
        StandardOpenOption.WRITE)) {
  fc.position(8);
  fc.write(bb1);
  // La ligne: Bonjour Wonsieur
```


Exemple (fin)

```
  // Lit les 5 premiers octets du fichier
  fc.position(0);
  fc.read(bb2);
  // Prépare bb2 pour l'écriture
  bb2.flip();
  // Ajoute les 5 octets à la fin du fichier
  fc.position(fc.size() - 1);
  fc.write(bb2);
  // La ligne : Bonjour WonsieurBonjo
}
```


Les flots

Flots (*streams*) - définition

- Les flots de données permettent d'échanger de données entre un programme et l'extérieur
- Le plus souvent un flot permet de transporter séquentiellement des données : les données sont transportées une par une (ou groupe par groupe), de la première à la dernière donnée

Flot - utilisation

- Le cycle d'utilisation de lecture ou écriture séquentielle d'un flot de données est le suivant :
  1) Ouvrir le flot
  2) Tant qu'il y a des données à lire (ou à écrire), lire (ou écrire) la donnée suivante dans le flot
  3) Fermer le flot

Sources ou destinations de flots

- Fichier
- *Socket* pour échanger des données sur un réseau
- URL (adresse Web)
- Données de grandes tailles dans une base de données (images, par exemple)
- *Pipe* entre 2 files d'exécution (*threads*)
- Tableau d'octets
- Chaîne de caractères
- etc...

Survol du paquetage **`java.io`**

Paquetage **`java.io`**

- Il contient la plupart des classes liées aux entrée-sorties
- Il prend en compte un grand nombre de flots :
  - 2 types de flots (octets et caractères)
  - différentes sources et destinations
  - « décorations » diverses
- Le grand nombre de classes de ce paquetage peut effrayer le débutant

2 types de flots

- Les flots d'octets servent à lire ou écrire des octets « bruts » qui représentent des données manipulées par un programme
- Les flots de caractères servent à lire ou écrire des données qui représentent des caractères lisibles par un homme, codés avec un certain codage (ISO 8859-1, UTF 8,…)

Types de classes

- Dans les 2 hiérarchies pour les flots d'octets et de caractères, on trouve :
  - des classes de base, qui sont associées à une source ou une destination « concrète » Exemple : **`FileReader`** pour lire un flot de caractères depuis un fichier
  - des classes qui « décorent » une autre classe Exemple : **`BufferedReader`** qui ajoute un *buffer* pour lire un flot de caractères

Décorations des flots

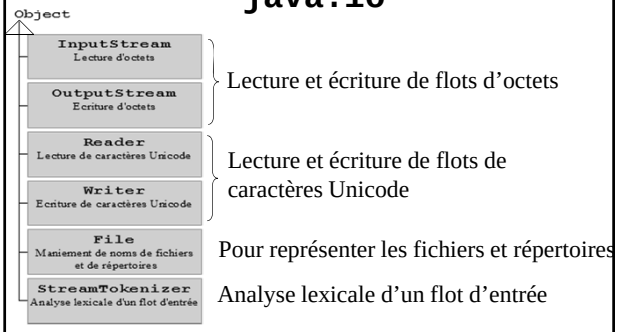
- Les fonctionnalités de base d'un flot sont la lecture ou l'écriture (méthodes **`read`** ou **`write`**)
- Selon les besoins, on peut lui ajouter d'autres fonctionnalités/décorations :
  - Utilisation d'un *buffer* pour réduire les lectures ou écritures « réelles »
  - Codage ou décodage des données manipulées
  - Compression ou décompression de ces données
  - etc...

Exemple de décoration

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

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

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



Lecture et écriture de flots d'octets

Lecture et écriture de flots de caractères Unicode

Pour représenter les fichiers et répertoires

Analyse lexicale d'un flot d'entrée

Sources et destinations concrètes

- Fichiers :
  - **`File`**{**`In`**|**`Out`**}**`putStream`**
  - **`File`**{**`Reader`**|**`Writer`**}
- Tableaux
  - **`ByteArray`**{**`In`**|**`Out`**}**`putStream`**
  - **`CharArray`**{**`Reader`**|**`Writer`**}**`}`**
- Chaînes de caractères
  - **`String`**{**`Reader`**|**`Writer`**}**`}`**

Lit ou écrit un Buffer d'octets ou de **`char`** avec un flot

Lit ou écrit une **`String`** avec un flot

Décorateurs (ou filtres)

- Pour bufférisers les entrées-sorties :
  - **`Buffered`**{**`In`**|**`Out`**}**`putStream`**
  - **`Buffered`**{**`Reader`**|**`Writer`**}**`}`**
- Pour permettre lecture et écriture des types primitifs sous une forme binaire :
  - **`Data`**{**`In`**|**`Out`**}**`putStream`**
- Pour compter les lignes lues :
  - **`LineNumberReader`**

Décorateurs (suite)

- Pour écrire dans un flot tous les types de données sous forme de chaînes de caractères :
  - **`PrintStream`**
  - **`PrintWriter`**
- Pour permettre de replacer un caractère lu dans le flot :
  - **`PushbackInputStream`**
  - **`PushbackReader`**

Lecture et écriture de flots d'octets

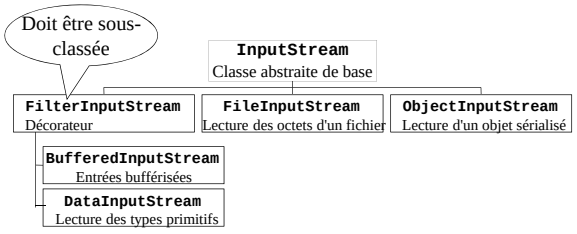
Classe pour entrée	Classe pour sortie	Fonctions fournies
`InputStream`	`OutputStream`	Classes abstraites de base pour les lecture et écriture d'un flot de données
`FilterInputStream`	`FilterOutputStream`	Classe mère des classes qui ajoutent des fonctionnalités à Input/OutputStream
`BufferedInputStream`	`BufferedOutputStream`	Lecture et écriture avec buffer
`DataInputStream`	`DataOutputStream`	Lecture et écriture des types primitifs
`FileInputStream`	`FileOutputStream`	Lecture et écriture d'un fichier
	`PrintStream`	Possède les méthodes « print() », « println() » utilisées par System.out

Lecture et écriture de flots de caractères

Classe pour entrée	Classe pour sortie	Fonctions fournies
`Reader`	`Writer`	Classes abstraites de base
`InputStreamReader`	`OutputStreamReader`	Ponts entre les flots d'octets et les flots de caractères
`FileReader`	`FileWriter`	Lecture et écriture de caractères à partir des octets d'un fichier (codage par défaut)
`BufferedReader`	`BufferedWriter`	Lecture et écriture avec buffer
	`PrintWriter`	Possède les méthodes « print() » et « println() »

Lecture de flots d'octets

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



Classe **`InputStream`**

- Classe abstraite
- C'est la racine des classes liées à la lecture d'octets depuis un flot de données
- « Interface » selon laquelle sont vues toutes les classes de flot qui lisent des octets (cf. modèle de conception « décorateur »)
- Elle possède un constructeur sans paramètre

Méthodes de la classe **`InputStream`**

- Interface publique de cette classe :

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

synchronized void mark(int nbOctetsLimite)
synchronized void reset() throws IOException
public boolean markSupported()
```


Description des méthodes

- **`int read()`**
  - renvoie l'octet lu dans le flot (sous forme d'un entier 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 d'une exception (comme toutes les autres méthodes **`read`**)
  - abstraite

Description des méthodes

- **`int read(byte[] b)`**
  - essaie de lire assez d'octets pour remplir le tableau **`b`**
  - renvoie le nombre d'octets réellement lus (elle est débloquée par la disponibilité d'au moins un octet), ou -1 si elle a rencontré la fin du flot
  - implémentée en utilisant la méthode **`read()`** (à redéfinir dans les classes filles pour de meilleures performances)
- Il est tout à fait possible que la méthode retourne avant d'avoir rempli tout le tableau **`b`**

Description des méthodes (2)

- **`int read(byte[] b, int début, int nb)`**
  - lit **`nb`** octets et les place dans le tableau **`b`** à partir de l'indice **`début`** (de **`début`** à **`début`** + **`nb`** - 1)
  - renvoie le nombre d'octets lus, ou -1 si elle a rencontré la fin du flot
- **`long skip(long n)`**
  - saute **`n`** octets dans le flot
  - renvoie le nombre d'octets réellement sautés
- **`int available()`**
  - renvoie le nombre d'octets prêts à être lus

Toutes ces méthodes sont à redéfinir dans les classes filles (pour de meilleures performances)

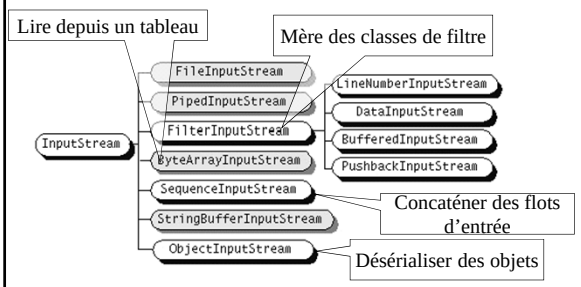
Description des méthodes (3)

- **`boolean markSupported()`**
  - indique si le flot supporte la notion de marque pour revenir en arrière durant la lecture
- **`void mark(int readlimit)`**
  - marque la position actuelle pour un retour ultérieur éventuel à cette position avec **`reset`**
  - **`readlimit`** indique le nombre d'octets lus après lequel la marque peut être « oubliée »
- **`void reset()`**
  - positionne le flot à la dernière marque

Description des méthodes (4)

- **`void close()`**
  - ferme le flot. Il est important de fermer les flots qui ne sont plus utilisés (sauf exceptions signalées dans la javadoc). En effet, des données du flot peuvent être perdues si le flot n'est pas fermé. De plus les flots ouverts sont souvent des ressources qu'il faut économiser.

Sous-classes de **`InputStream`**



En grisé : classes associées à des sources et destination « concrètes »

`readFully(byte[] b)`

- Méthode définie dans l'interface **`java.io.DataInput`** qu'implémente **`DataInputStream`**
- Bloque jusqu'au remplissage du tableau **`b`**, ou la rencontre de la fin de fichier (**`EOFException`** renvoyée), ou une erreur d'entrée/sortie (**`IOException`** renvoyée)
- La variante **`readFully(byte[] b, int debut, int fin)`** permet de ne remplir qu'une partie du tableau

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

On peut décorer un décorateur

- L'exemple du transparent suivant montre qu'un décorateur peut décorer un autre décorateur
- C'est possible parce que, selon le pattern décorateur,
  - le décorateur décore un **`InputStream`**
  - et que le décorateur et le décoré « *sont-des* » **`InputStream`** (par héritage)

Classes de l'exemple

- L'exemple utilise
  - **`FileInputStream`** : classe de base pour la lecture d'un fichier ; décorée par un
  - **`BufferedInputStream`** : décorateur qui ajoute un buffer pour la lecture du flot ; décoré par un
  - **`DataInputStream`** : décorateur qui décode les types primitifs Java codés dans un format standard, indépendant du système

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();
```

Codage UTF-8 (*Unicode Text Format*) pour les **`String`**

A mettre dans un **`finally`** (voir section « Exceptions » plus loin dans ce cours)

Variante de l'exemple

- En fait, comme on n'utilisera que le flot décoré **`dis`**, on n'a pas besoin des variables intermédiaires et on écrira :

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


Intérêt du pattern décorateur

- L'héritage permet aussi d'ajouter des fonctionnalités
- Quand choisir le pattern décorateur plutôt que l'héritage ?

Intérêt du pattern décorateur

- Il est utile quand un objet de base peut être décoré de multiples façons
- Si on utilisait l'héritage, on aurait de nombreuses classes, chacune 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

Les filtres

- Dans le JDK, les décorateurs sont appelés filtres
- On va étudier l'implémentation du pattern décorateur avec ces filtres
- Les décorateurs de flots d'entrée héritent de la classe **`FilterInputStream`**

Constructeur des filtres

- Le constructeur **`protected`** de **`FilterInputStream`** garde le flot à décorer dans une variable d'instance **`in`** (**`protected`**, de type **`InputStream`**) :

```
FilterInputStream(InputStream in) {
  this.in = in;
}
```

- Ce constructeur est appelé par les constructeurs des classes de décorateurs; par exemple :

```
BufferedInputStream(InputStream in) {
  super(in);
  . . .
```


Mécanisme des filtres

- Quand on demande au filtre de lire une donnée,
  - le filtre fait son traitement (par exemple, chercher s'il a déjà la donnée dans son buffer)
  - fait appel à **`in`** s'il a besoin du flot qu'il décore (par exemple s'il a besoin d'une lecture réelle)
- **`in`** peut lui-même être un filtre car les classes des filtres sont des sous-classes de **`InputStream`** (*design pattern* décorateur)

Lire les octets d'un fichier

- Pour lire un fichier qui contient des octets qu'on ne peut lire sous forme de types Java particuliers (images, vidéo, etc…) :

```
File f = new File("fichier");
int tailleFichier = (int)f.length();
byte[] donnees = new byte[tailleFichier];
DataInputStream dis =
  new DataInputStream(
    new FileInputStream(f));
dis.readFully(donnees);
dis.close();
```

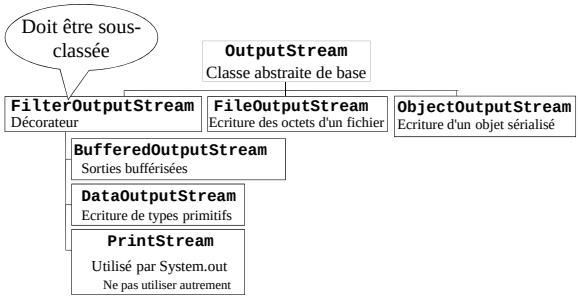
A mettre dans un **`finally`** (voir section « Exceptions » plus loin dans ce cours)

Fermeture des filtres

- La fermeture d'un filtre du JDK ferme le flot qu'il décore
- Dans l'exemple du transparent précédent, la fermeture de **`dis`** suffit pour fermer les flots qu'il décore

Écriture de flots d'octets

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



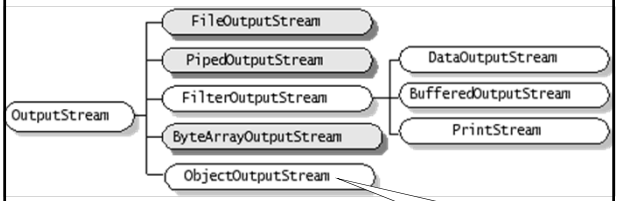
Classe **`OutputStream`**

- Interface publique de cette classe (ajouter **`throws IOException`** à toutes les méthodes) :

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

- Remarque : avec la méthode **`write(int b)`**, seul l'octet de poids faible de **`b`** est écrit dans le flot

Sous-classes de **`OutputStream`**



Particularités de **`PrintStream`**

- Cette classe possède les 2 méthodes **`print()`** et **`println()`** qui écrivent tous les types de données sous forme de chaînes de caractères
- Aucune des méthodes de **`PrintStream`** ne lève d'exception ; on peut savoir s'il y a eu une erreur en appelant la méthode **`checkError()`**
- Attention, **`println()`** n'effectue un **`flush()`** (vidage des buffers) que si le **`PrintStream`** a été créé avec le paramètre « *autoflush* »

Utilisation de **`ByteArrayOutputStream`**

- Utile lorsque l'on veut ranger des octets dans un tableau **`octets`**, sans connaître au départ le nombre d'octets :

```
ByteArrayOutputStream out =
    new ByteArrayOutputStream();
// On envoie des octets dans le flot
int b;
while ((b = autreValeur()) > 0) {
    out.write(b);
}
// On récupère les octets dans un tableau
byte[] octets = out.toByteArray();
```


Utilisation de **`ByteArrayOutputStream`**

- On peut aussi récupérer les octets sous la forme d'une **`String`**
- La méthode **`toString()`** de **`ByteArrayOutputStream`** utilise pour cela le codage par défaut des caractères
- On peut choisir un autre codage avec la méthode **`toString(String codage)`** :

```
byte[] octets = out.toString("UTF8");
```


Écrire des types primitifs dans un fichier

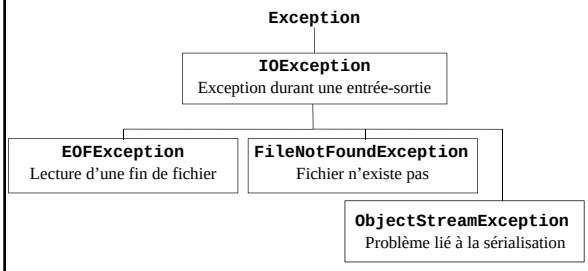
```
DataOutputStream dos =
  new DataOutputStream(
    new BufferedOutputStream(
      new FileOutputStream("fichier")));
dos.writeDouble(12.5);
dos.writeUTF("Dupond");
dos.writeInt(1254);
dos.close();
```

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

- Un traitement des exceptions correct est indispensable lors des traitements des entrées-sorties
- Nombreuses variantes dans le traitement des exceptions suivant ce que l'on veut faire
- Attention, pour simplifier leur lecture, quelques exemples de ce cours, ne comportent pas le traitement des exceptions
- Les 3 transparents qui suivent sont des exemples complets de traitement des exceptions
- Par manque de place, la lecture est effectuée sans buffer ; il faudrait décorer avec un **`BufferedInputStream`**

Lire des types primitifs dans une boucle ; traitement des exceptions

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

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

Remarquez l'emplacement des blocs **`catch`** et **`finally`**

Traitement des exceptions ; variante

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

1 seul bloc **`try`**

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) { . . . }
```

Le JDK 7 fournit le try-avec-ressources
(voir cours sur les exceptions)

Cas où les exceptions ne sont pas traitées mais renvoyées par la méthode

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

Peut-on/Doit-on enlever cette ligne ?

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

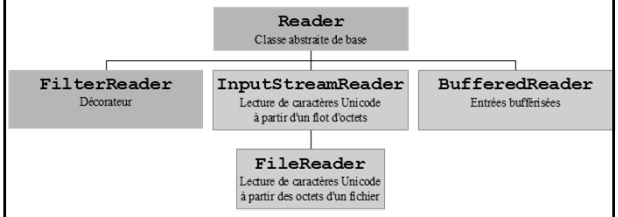
```
public void lire(String fichier) throws IOException {
  try (
     DataInputStream dis =
       new DataInputStream(new FileInputStream(fichier))
    ) {
    while (true) {
      double d = dis.readDouble();
      . . .
    }
  }
  catch(EOFException e) {}
}
```


Lecture d'un flot de caractères

Classes de base

- La classe **`Reader`** lit des caractères (**`char`**) dans un flot
- **`Writer`** envoie des caractères dans un flot
- Ces 2 classes sont abstraites

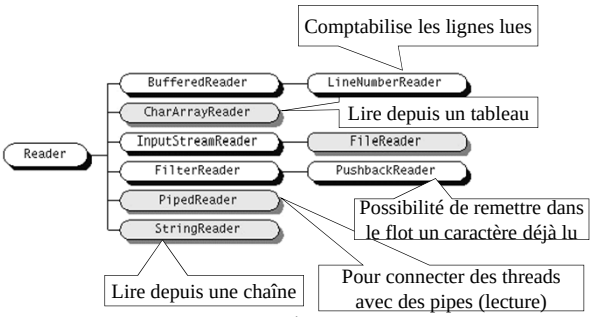
Hiérarchie des principales classes de lecture d'un flot de caractères



Méthodes publiques de la classe **Reader**

```
int read() throws IOException
int read(char[] b) throws IOException
abstract int read(char[] b, int début, int nb)
     throws IOException
long skip(long n) throws IOException
boolean ready() throws IOException
abstract void close() throws IOException
synchronized void mark(int nbOctetsLimite)
synchronized void reset() throws IOException
boolean markSupported()
```


Sous-classes de **Reader**



Lecture d'un flot composé de lignes de texte

- On utilise la classe **BufferedReader** qui comprend la méthode **readLine()**

Séparateurs des données

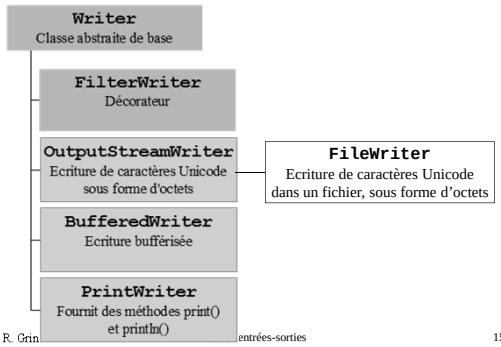
- Pour les flots d'octets, il suffit de relire les données dans l'ordre dans lequel elles ont été écrites
- Pour les flots de caractères, on doit explicitement mettre des séparateurs entre les données ; par exemple, pour distinguer un nom d'un prénom

Relire des données avec séparateurs

- Le plus simple est d'utiliser la méthode **split** de la classe **String** pour décomposer les lignes du flot

Écriture dans un flot de caractères

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



Méthodes publiques de la classe **`Writer`**

```
void write(int c) throws IOException
void write(char[] b) throws IOException
abstract void write(char[] b, int début, int
  nb) throws IOException
void write(String s)
void write(String s, int décalage, int
  longueur)
abstract void flush(long n) throws IOException
abstract void close() throws IOException
```

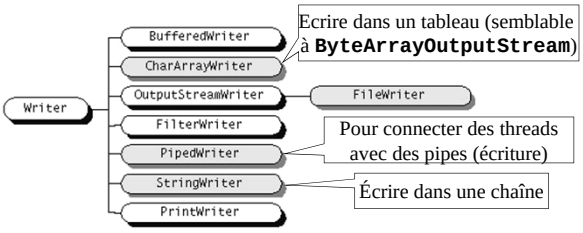

Méthodes publiques de la classe **`Writer`**

- (Ajouter **`throws IOException`** à toutes les méthodes)

```
void write(int c)
void write(char[] b)
abstract void write(char[] b, int déb, int nb)
void write(String s)
void write(String s, int déb, int nb)
abstract void flush(long n)
abstract void close()
```

N'écrit que les 2 octets de poids faibles

Sous-classes de **`Writer`**



`PrintWriter`

- Cette classe est un décorateur pour un **`OutputStream`** ou un **`Writer`**
- Elle contient les méthodes **`print`**, **`println`** ou **`printf`** qui permettent d'écrire dans une flot comme si on affichait sur l'écran (le fameux **`System.out.println`**)
- Ces méthodes ne lancent jamais d'exception ; la méthode **`boolean checkError()`** renvoie **`true`** s'il y a eu une exception en interne

Constructeurs de **`PrintWriter`**

- Le JDK 5 a offert des facilités pour créer directement un **`PrintWriter`** qui écrit dans un fichier dont on donne le nom, sans créer explicitement les flots sous-jacents décorés
- Un buffer est utilisé mais les **`println`** ne provoquent pas de *flush* automatique

Constructeurs de **`PrintWriter`**

- **`PrintWriter(String`** ***`nomFichier`*****`[, String`** ***`nomCharset`*****`])`**
  un 2ème paramètre permet de donner le nom d'un autre **`Charset`** que celui par défaut
- **`PrintWriter(File`** ***`fichier`*** **`[, String`** ***`nomCharset`*****`])`**

Constructeurs de **`PrintWriter`**

- Avant le JDK 5, il fallait passer un **`OutputStream`** ou un **`Writer`** en paramètre
- On doit encore utiliser cette décoration explicite (voir exemples plus loin dans cette section)
  - si on veut ne pas écraser un fichier existant (mais écrire à la fin du fichier)
  - si on veut un flush automatique à chaque **`println`**

Séparer les lignes

- La façon de séparer les lignes dépend du système d'exploitation
- Pour être portable utiliser
  - **`println`** de **`PrintWriter`** (le plus simple)
  - **`writeLine`** ou **`newLine`** de **`BufferedWriter`**
  - ou la propriété système **`line.separator`** (**`System.getProperty("line.separator")`**)
- Ne pas utiliser le caractère **`\n`** qui ne convient pas, par exemple, pour Windows

Lecture et écriture de caractères dans des fichiers – codage des caractères

Codage

- En Java les caractères sont codés en Unicode
- Ce n'est souvent pas le cas sur les périphériques source ou destination des flots (le plus souvent ASCII étendu ISO 8859-1 pour les français, mais le codage UTF-8 tend à se généraliser)
- Des classes spéciales permettent de faire les traductions entre le codage Unicode et un autre codage
- Un codage par défaut est automatiquement installé par le JDK, conformément à la locale

Codages supportés par Java

- Liste à l'adresse (intl avec la lettre l à la fin) http://java.sun.com/j2se/1.5.0/docs/guide/intl/encoding.doc.html
- Par exemple, le codage par défaut pour les langues d'Europe de l'Ouest est le codage ISO 8859-1 qui est représenté en Java par le nom ISO8859_1 (il semble que ISO-8859-1 est aussi accepté par java.nio)
- On peut aussi trouver les codages de noms UTF-8, UTF-16, et de très nombreux autres codages

Ponts entre les flots de caractères et les flots d'octets

- **`InputStreamReader`** et **`OutputStreamWriter`** sont des classes filles de **`Reader`** et **`Writer`**
- **`InputStreamReader`** lit des caractères dans un flot ; ces caractères, codés dans le flot suivant un codage particulier, sont décodés en caractères Unicode
- **`OutputStreamWriter`** écrit des caractères Unicode en les codant sous forme d'octets en utilisant un codage particulier

Ponts entre les flots de caractères et les flots d'octets

- Leur constructeur prend en paramètre un flot d'octets ; par exemple,

```
public InputStreamReader(InputStream in)
```

- Les octets sont lus dans le flot **`in`** et sont décodés en caractères Unicode par le codage associé à **`InputStreamReader`**
- Le codage par défaut est défini par la locale (ISO-8859-1 en France)

Ponts entre les flots de caractères et les flots d'octets

- On peut préciser un codage particulier en paramètre du constructeur (idem pour **`OutputStreamWriter`**) si on ne veut pas le codage par défaut :

```
public InputStreamReader(InputStream in,
                         Charset cs)
```

- Exemple :

```
new InputStreamReader(
      in, Charset.forName("UTF-8"))
```


Lecture-écriture dans un fichier de texte

- **`File`**{**`Reader`**|**`Writer`**} sont des classes filles de **`InputStreamReader`** et **`OutputStreamWriter`**
- Elles permettent de lire et d'écrire des caractères Unicode dans un fichier, suivant le codage par défaut
- Utiliser leur classe mère si on veut un autre codage ; par exemple, utiliser un **`InputStreamReader`** avec le codage voulu, pour décorer un **`FileInputStream`**

Exemple

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


Travail avec un fichier composé de lignes de texte

- En lecture, on utilise la classe **`BufferedReader`** qui comprend la méthode **`readLine`**
- En écriture, on utilise la classe **`PrintWriter`** qui comprend les méthodes **`print`**, **`println`** et **`printf`**

Remarque

- Dans les 3 exemples de code suivants on n'attrape pas les **`IOException`**
- L'en-tête de la méthode qui contient le code devra donc comporter « **`throws IOException`** »
- Dans la pratique ça sera le plus souvent le cas car la méthode qui fait les entrées-sorties sait rarement comment réparer en cas de **`IOException`**

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();
}
```

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

Lire les lignes de texte - JDK 7

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


Écrire une ligne de texte (JDK 5)

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

Un buffer est automatiquement utilisé

Ne pas oublier la fermeture du flot !

Écrire une ligne de texte (JDK 1.4)

```
PrintWriter pw =
  new PrintWriter(
    new BufferedWriter(
      new FileWriter("fichier")),
    true);
String ligne;
. . .
pw.println(ligne);
. . .
```

Un paramètre optionnel permettrait d'ajouter à la fin du fichier

Si on veut un vidage des buffers après chaque **`println()`**

Un buffer n'était pas automatiquement utilisé

Ne pas oublier la fermeture du flot !

Recopier un fichier texte sur un autre

```
BufferedReader in = null;
BufferedWriter out = null;
try {
  in =
    new BufferedReader(new FileReader("f1"));
  out =
    new BufferedWriter(new FileWriter("f2"));
  int c; // code Unicode du caractère lu
  while ((c = in.read()) != -1)
    out.write(c);
}
finally {
  if (in != null) in.close();
  if (out != null) out.close();
}
```

Test de fin de fichier

Recopier un fichier texte – JDK 7

```
try (
  BufferedReader in =
    new BufferedReader(new FileReader("f1"));
  BufferedWriter out =
    new BufferedWriter(new FileWriter("f2"));
) {
  int c; // code Unicode du caractère lu
  while ((c = in.read()) != -1)
    out.write(c);
}
```


Changer de codage

```
BufferedReader reader =
  new BufferedReader(new InputStreamReader(
    source, codageSource));
PrintWriter writer =
  new PrintWriter(new OutputStreamWriter(
    destination, codageDestination));
String ligne;
while ((ligne = reader.readLine()) != null){
  System.out.println(ligne);
  writer.println(ligne);
}
```

Ne pas oublier la fermeture du flot !

Changer de codage

- On peut utiliser ce code avec, par exemple :

```
source =
  new FileInputStream(new File(...))
codageSource =
  Charset.forName("ISO-8859-1")
destination =
  new FileOutputStream(new File(...))
codageDestination =
  Charset.forName("UTF-8")
```


URL et URI

URI et URL

- Un URI (*Uniform Resource Identifier*) est un identificateur d'une ressource accessible localement ou sur Internet
- Un URL (*Uniform Resource Locator*) est un type d'URI qui identifie une ressource par son emplacement sur le réseau
- Il existe aussi un autre type d'URI, rarement utilisé, les URN (N pour *Name*) qui identifient une ressource par un nom, indépendant de son emplacement sur le réseau ; par exemple « urn:ietf:rfc:2141 »

URL

- Un URL peut être représenté par une chaîne de caractères du type ***`protocole:nomRessource`***
- Le format pour le nom de la ressource dépend du protocole
- Exemple avec nom absolu : **`http://deptinfo.unice.fr/~toto/index.html`**
- Exemple avec nom relatif : **`file:rep1/rep2`**

Le séparateur est toujours « **`/`** », pour tous les systèmes

Caractères d'un URL

- Un URL ou URI ne peut contenir qu'un sous-ensemble des caractères du code ASCII : lettres, chiffres et les caractères
  - _ . ! ~ * ' ,
- Les caractères
  ? & @ % / # ; : $ + =
  servent pour séparer les différentes parties d'un URL ; par exemple, **?** sert à séparer une adresse d'une chaîne passée en paramètre

Caractères spéciaux

- Les autres caractères sont encodés dans des octets ; chaque octet est représenté par un **%** suivi du code hexadécimal de l'octet
- Par exemple, un espace est représenté par **%20** (car le code hexadécimal de l'espace est 20 : 32 en décimal) ; pour faciliter l'usage des espaces, ils sont aussi encodés par un « **+** » (**+** est lui-même encodé par **%2B**)

Problèmes de codage

- Cette façon d'encoder certains caractères pose des problèmes en environnement hétérogène
- Ainsi « é » n'est pas codé de la même façon sur un Macintosh et sur un système Windows
- Il faut donc indiquer le codage des caractères que l'on veut utiliser ; il est conseillé d'utiliser au maximum UTF-8 pour éviter les problèmes de portabilité

Problèmes de codage

- Les classes **URLEncoder** et **URLDecoder** du paquetage **java.net** sont des méthodes **static** qui effectuent le codage/décodage :
  - **URLEncoder.encode(String, String)** transforme tous les caractères interdits en caractères autorisés
  - **URLDecoder.decode(String, String)** fait l'inverse
- Le 2ème paramètre indique le codage des caractères

Exemple

```
URL url = getclass().getResource(
"/un répertoire/fichier.txt");
System.out.println(url);
System.out.println(
  URLDecoder.decode(url.toString(),
                    "UTF-8"));
```

- affiche

```
file:/…/un%20r%c3%a9pertoire/fichier.txt
file:/…/un répertoire/fichier.txt
```


URL sous Windows

- Ne pas utiliser « **** » mais « **/** » comme séparateur dans les noms de fichiers
- Plusieurs formats sont acceptés pour un même emplacement
- Noms absolus (le 1er est préférable) :
  file:/C:/autoexec.bat
  file:C:/autoexec.bat
- Noms relatifs :
  file:C:truc.txt
  file:truc.txt (si C est le disque courant)

Principaux protocoles

- **`http`** pour le protocole HTTP (pages HTML)
- **`file`** pour les fichiers locaux (**`file:chemin`**) ou les fichiers sur une autre machine (**`file://hote/chemin`**)
- **`ftp`** pour FTP (transfert de fichiers)
- **`telnet`** pour une connexion par *telnet*
- **`news`** pour les *news*

Adresse Web

- Format des adresses http :
  `http://`*`machine`*`[:`*`port`*`]/`*`cheminPage`*`[#`*`ancre`*`]`
- Exemples
  `http://dept.unice.fr/~toto/index.html`
  `http://dept.unice.fr:8080/~toto/index.html`
  `http://dept.unice.fr/~grin/index.html#intro`

Classe **`URL`**

- La classe **`java.net.URL`** représente un URL
- Elle fournit de nombreux constructeurs
- Si la **`String`** passée à un constructeur ne correspond pas à la syntaxe des URL, le constructeur lance l'exception contrôlée **`java.net.MalformedURLException`**, fille de **`IOException`**

Constructeurs de la classe **`URL`**

- On peut créer un URL avec une **`String`**, ou à partir des éléments de base (machine, port, etc.) passés comme des **`String`**
- On peut aussi créer un URL en passant une adresse relative à un URL (le contexte) ; par exemple, si url correspond à l'URL http://deptinfo.unice.fr/tp/tp1/index.html, **`new URL(url, "../tp2/index.html")`** correspond à l'URL http://deptinfo.unice.fr/tp/tp2/index.html

Méthode de la classe **`URL`**

- On peut extraire les éléments de base à partir d'un URL (**`getPort`**, **`getHost`**, etc.)
- Les données associées à l'URL peuvent être obtenues par les méthodes **`openConnection`** et **`openStream`** ou par la méthode **`getContent`**

Lire le code HTML d'une page Web

- La classe **`URL`** fournit la méthode **`InputStream openStream() throws IOException`** qui permet de lire le contenu d'un URL :

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

URL relatif dans une applet

- Obtenir l'URL d'un fichier placé dans le même répertoire que le document HTML contenant une *applet* (**`getCodeBase`** pour une position relative à la classe de l'applet) :

```
URL urlDoc = getDocumentBase();
String nomFichier = getParameter("fichier");
try {
  urlFichier = new URL(urlDoc, nomFichier);
}
catch(MalformedURLException e) {
  . . .
}
```


Classe **`URLConnection`**

- La méthode **`openConnection()`** de la classe URL fournit une instance de **`URLConnection`**
- La classe **`URLConnection`** permet d'obtenir des informations sur l'URL (type, codage, date de dernière modification, etc.)
- On peut aussi obtenir un flot en lecture ou en écriture (**`getInputStream`**/**`getOuputStream`**) vers l'URL (si le protocole le permet)

Classe **`URI`**

- La classe **`java.net.URI`** contient (entre autres) des méthodes **`resolve`** et **`relativize`** qui facilitent le passage entre noms relatifs par rapport à un URI de base, et noms absolus des fichiers
- Elle permet aussi d'obtenir un **`Path`** à partir d'un **`URL`** par la méthode **`Paths.get(URI)`**:
  `Path path = Paths.get(url.toURI());`

`URI` – **`URL`** – **`Path`**

- Il est facile de passer d'une classe à l'autre pour profiter des fonctionnalités de chacune
- Passer de **`URL`** à **`URI`** : **`toURI()`** de la classe **`URL`**
- Passer de **`URI`** à **`URL`** : **`toURL()`** de la classe **`URI`**
- Passer de **`URI`** à **`Path`** (et donc de **`URL`** à **`Path`** avec **`toURI()`** avant) : **`Paths.get(URI)`**

Classe **`URI`**

- Les caractères interdits sont automatiquement traités si on passe de **`URI`** à **`URL`** :

```
URI uri = new URI(
  "http", "//m1.com/un url", null);
URL url = uri.toURL();
// Affiche « /un url »
System.out.println(uri.getPath());
// Affiche « /un%20url »
System.out.println(url.getPath());
```

Pour l'URL, l'espace sera remplacé par **`%20`**

Obtenir un nom de fichier à partir d'un URL

- Il n'est pas évident d'obtenir un nom de fichier à partir d'un **`URI`** (ou d'un **`URL`**) : extraction du chemin du fichier, puis décodage de ce nom pour enlever les caractères interdits dans un URL ; de plus il faut tenir compte du séparateur dans les noms de fichier, qui dépend du système d'exploitation
- Il faut utiliser la méthode **`Paths.get(URI)`**

`Paths.get(URI)`

- Cette méthode permet d'obtenir un **`Path`** à partir d'un **`URI`**
- Si on veut le nom du fichier correspondant, il suffit d'utiliser la méthode **`toString()`** de **`Path`**

Exemple : nom du répertoire qui contient le jar exécutable

- Si l'application est dans un jar, il peut être utile de connaître le nom du répertoire qui contient le jar (par exemple pour voir s'il contient des fichiers de configuration) :

```
URL urlRep =
  this.getClass().getResource("/");
Path rep = Paths.get(urlRep.toURI());
String nomRep = rep.toString();
```


Noms de fichiers, ressources

Un problème fréquent

- Un projet fonctionne dans l'environnement de développement mais ne fonctionne plus dès que l'application est distribuée sous la forme d'un fichier jar
- En effet les images ou les fichiers divers utilisés par l'application ne sont alors plus trouvés par l'application
- La solution est d'utiliser des noms de ressource et pas des noms de fichier pour désigner les images ou les fichiers divers

Le problème avec les noms de fichiers

- Certaines classes étudiées dans cette partie du cours (**`Files`** en particulier) désignent un fichier par son nom relatif ou absolu
- Si on utilise un nom absolu, il faudra recompiler l'application dès que le fichier changera de place
- L'utilisation de noms relatifs pose aussi des problèmes
- De plus le fichier peut se trouver dans un fichier jar

Le problème avec les noms relatifs

- Les noms relatifs sont relatifs au répertoire courant (propriété système **`user.dir`**)
- Le répertoire courant est généralement le répertoire dans lequel la JVM a été lancée (commande java)
- Mais on ne connaît pas à l'avance ce répertoire
- De plus, le répertoire courant est très difficile à maîtriser dans certains environnements complexes d'exécution (EJB par exemple)

Une meilleure solution

- Il est préférable d'utiliser un nom de ressource pour désigner le fichier et d'utiliser les méthodes de la classe **`Class`**
  `URL getResource(String nom)`
  ou
  `InputStream`
  `getResourceAsStream(String nom)`
- Le nom de la ressource passé en paramètre à ces 2 méthodes indique l'endroit où la ressource sera recherchée (voir transparents suivants)

Nom d'une ressource

- Cet endroit dépend de la façon dont la classe a été chargée en mémoire
- Le plus souvent (voir cours sur l'interface avec le système),
  - si le nom est relatif, il est relatif au répertoire où se trouve le fichier .class qui contient le code
  - si le nom est absolu (commence par « / »), il est *relatif* au *classpath*

Lire le contenu d'un fichier ressource

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

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

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

Cas d'une méthode **`static`**

- Le code précédent ne fonctionnera pas dans une méthode **`static`** à cause de « **`this.getClass()`** »
- En ce cas, il faut remplacer « **`this.getClass()`** » par l'instance d'une des classes de l'application (la classe qui contient le code ; pour faire simple, appelons-la « **`p.C1`** ») :

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


Lire un fichier ressource texte

- Si la ressource contient du texte, il faut utiliser **`InputStreamReader`** pour obtenir un **`Reader`** ; par exemple (**`in`** obtenu par **`getResourceAsStream`**) :

```
BufferedReader br =
  new BufferedReader(
    new InputStreamReader(in));
String ligne;
while ((ligne = br.readLine()) != null) {
```

- On peut spécifier le codage des caractères de la ressource si ça n'est pas le codage par défaut :

```
new InputStreamReader(
      in, Charset.forName("UTF-8"))
```


Écrire dans une ressource (1/2)

- Si on veut écrire dans un fichier qui existe déjà en utilisant un nom de ressource pour désigner le fichier, il est préférable de passer par la classe URL (si le fichier n'existe pas déjà, il faut s'arranger autrement…)
- On récupère d'abord l'URL du fichier par `url = getClass().getResource("/fichier");`
- Ensuite il faut ouvrir un flot en écriture sur l'URL

Écrire dans une ressource (2/2)

- Pour ouvrir une ressource en écriture, le plus simple est de récupérer le nom du fichier à partir de l'**`URL`** (voir transparent suivant si le nom contient des caractères spéciaux)
- Exemple (pour un fichier au format texte) :

```
URL url =
  getClass().getResource("/fichier");
PrintWriter pw = new PrintWriter(
  url.getPath());
```


Chemin de la ressource

- Il peut parfois être utile d'avoir le chemin du fichier dans l'arborescence des fichiers ; voici un exemple de code pour l'obtenir :

```
URL urlRessource = this.getClass()
  .getResource("/rep/fich");
String nomFichier = urlRessource.getPath();
// Voir section URL pour utilité
// de URLDecoder.decode
nomFichier = URLDecoder
  .decode(nomFichier, "ISO-8859-1");
```

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

Cas particulier d'un jar

- Si l'application est distribuée dans un jar et lancé par l'option **`-jar`** de java, un chemin absolu désignera un emplacement dans le jar relatif à la racine du jar

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

Sérialisation

Définition

- Sérialiser un objet c'est transformer l'état (les valeurs des variables d'instance) de l'objet en une suite d'octets
- On peut ainsi conserver l'état de l'objet pour le retrouver ensuite et reconstruire un autre objet avec le même état

Qu'est-ce qui est sérialisé ?

- Les valeurs des variables d'instance (pas des variables de classe) des instances sérialisées
- Des informations sur les classes des objets sérialisés ; en particulier :
  - nom de la classe
  - noms, types, modificateurs des variables à sauvegarder
  - des informations qui permettent de savoir si une classe a été modifiée entre la sérialisation et la désérialisation
- Mais le code des classes n'est pas sérialisé !

Utilisation de la sérialisation

- Conserver un objet dans un fichier ou une base de données pour le récupérer plus tard
- Conserver la configuration d'un composant, pour pouvoir le réutiliser plus tard dans une application (*JavaBean*)
- Transmettre les arguments (de types non primitifs) d'une méthode appelée sur un objet distant (RMI) : l'argument est sérialisé, les octets sont transmis sur le réseau et l'objet est reconstruit sur la machine distante

`ObjectOutputStream`

- Classe qui permet de sérialiser des objets ou des types primitifs
- Les méthodes **`writeObject`**, **`writeInt`**, **`writeDouble`**,… peuvent lancer une **`IOException`**

`ObjectOutputStream`

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

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

`ObjectInputStream`

- Classe qui permet de désérialiser des objets ou des types primitifs
- Les méthodes **`readObject`**, **`readInt`**, **`readDouble`**,… peuvent lancer une **`IOException`** ou une **`EOFException`**
- De plus la méthode **`readObject`** peut lancer divers autres IOException ou une **`ClassNotFoundException`**

`ObjectInputStream`

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

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

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

Fin de fichier

- Si on lit par boucle plusieurs objets sérialisés, on doit tester la fin de fichier avec **`EOFException`** (le test de la valeur **`null`** renvoyée par **`readObject`** ne marche pas)

Remarque importante

- Si on sérialise un objet et qu'on le désérialise, on obtient un nouvel objet
- Ce nouvel objet a le même état que l'objet d'origine (sauf si on a fait un traitement particulier pour le sérialiser) mais ça n'est pas l'objet d'origine

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

Classes **`Serializable`**

- La plupart des classes du JDK sont sérialisables
- Certaines classes ne peuvent pas être sérialisées (**`InputStream`** par exemple) ; d'autres ne doivent pas l'être (par sécurité)

Classes **`Serializable`**

- Le plus souvent rendre une classe sérialisable ne nécessite l'écriture d'aucune ligne de code
- Si la classe ne contient que des champs de types primitifs ou d'un type qui implémente **`Serializable`**, ou des tableaux de ces types, il suffit d'ajouter « **`implements Serializable`** » dans l'en-tête de la classe

Ne pas sérialiser une variable : **`transient`**

- Si on ne veut pas qu'une variable d'instance soit sérialisée, on la déclare **`transient`** : **`private transient int val;`**
- Quand l'objet sera désérialisé, la valeur de cette variable devra être recalculée s'il en est besoin (dans une méthode **`readObject`** privée ; voir transparents suivant), sinon elle recevra la valeur par défaut de son type

Sérialisation spéciale

- On peut choisir sa propre façon de sérialiser les objets d'une classe
- Dans la classe, on doit alors écrire les 2 méthodes qui doivent être **`private`** et lancer **`IOException`**
  `void writeObject(ObjectOutputStream oos)`
  `void readObject(ObjectInputStream ois)`
- Attention, ce ne sont pas les méthodes de même nom des classes **`Object`**{**`Out`**|**`In`**}**`putStream`**
- Curieux des méthodes **`private`** qui sont utilisées de l'extérieur, non ?

Sérialisation spéciale

- Les 2 méthodes **`readObject`** et **`writeObject`** n'ont à s'occuper que des variables de la classe
- Elles ne doivent pas s'occuper des classes ancêtres

Sérialisation spéciale (2)

- Ces 2 méthodes peuvent utiliser les méthodes **`default`**{**`Read`**|**`Write`**}**`Object()`** Ces méthodes font une (dé)sérialisation normale
- Si le traitement spécial ne concerne que des variables **`transient`**, on utilise ces 2 méthodes et il ne reste plus alors qu'à traiter d'une façon spéciale les variables **`transient`**

Exemple de sérialisation spéciale

```
private void writeObject(ObjectOutputStream oos)
  throws IOException {
  temp = motDePasse.clone();
  motDePasse = crypt(motDePasse);
  oos.defaultWriteObject();
  motDePasse = temp;
}
```

Le mot de passe est encrypté avant d'être sérialisé

```
private void readObject(ObjectInputStream ois)
  throws IOException, ClassNotFoundException {
  ois.defaultReadObject();
  motDePasse = deCrypt(motDePasse);
}
```


Empêcher la sérialisation

- Pour écrire une classe non sérialisable alors qu'elle hérite d'une classe sérialisable, il suffit de lui ajouter des méthodes **`readObject`** et **`writeObject`** qui lancent une **`NotSerializableException`**

Classe mère non sérialisable

- Une sous-classe d'une classe non sérialisable peut être sérialisable
- Dans ce cas c'est le rôle de la classe fille de donner des valeurs aux variables de la classe mère non sérialisable (avec les méthodes **`private`** {**`read`**|**`write`**}**`Object`**)
- La classe mère non sérialisable doit avoir un constructeur sans paramètre qui sera utilisé par la classe fille

Sérialisation spéciale avec `writeReplace` et `readResolve`

- Ces 2 méthodes n'ont pas de paramètre et renvoie un **`Object`**
- **`writeReplace`** est utilisée pendant la sérialisation et **`readResolve`** pendant la désérialisation, si elles sont accessibles (cf **`public`**, **`private`**,…)
- Si une classe contient ces méthodes, elles sont utilisées pour remplacer l'objet qui va être sérialisé/désérialisé par l'objet qu'elles renvoient

Utilisation de `writeReplace` et `readResolve`

- Les classes « d'énumération de constantes » utilisent ces méthodes (surtout **`readResolve`**) pour pouvoir conserver l'identité d'une constante
- En effet, si on sérialise une constante, la valeur désérialisée n'est pas égale (**`==`**) à la valeur sérialisée
- Depuis le JDK 1.5 il est plus simple et préférable d'utiliser **`enum`** pour représenter une énumération et ces 2 méthodes ne sont alors plus utiles

Sérialisation spéciale avec `writeReplace` et `readResolve`

- Le principe est de conserver les constantes dans un tableau **`static`**
- Pour sérialiser une des constantes, on sérialise l'indice du tableau qui permet de répérer la constante et on utilise cet indice au moment de la désérialisation pour renvoyer la bonne constante

Sérialisation et modification des classes (1)

- Si on modifie une classe, on ne peut relire les instances sérialisées avec l'ancienne version de la classe (**`java.io.InvalidClassException`**)
- On peut tout de même récupérer les valeurs des variables qui n'ont pas changé en ajoutant
  `private static final long serialVersionUID = xxxxxxxxL;`
  comme variable de classe, en lui donnant la valeur calculée sur l'ancienne version de la classe (utiliser pour cela l'outil ***`serialver`***)

Sérialisation et modification des classes (2)

- Pour cela il est souvent conseillé de donner une valeur quelconque à la variable **`serialVersionUID`** des classes que l'on écrit
- Ainsi il sera possible (si les modifications effectuées dans les nouvelles versions le permettent) de récupérer les instances sérialisées avec des anciennes versions de la classe

Sérialisation et modification des classes

- Pour les cas plus complexes, on peut utiliser aussi la méthode **`get(String, )`** (surchargée pour le 2ème paramètre avec tous les types primitifs et **`Object`**) de la classe interne **`ObjectInputStream.GetField`**
- Cette méthode permet de récupérer la valeur d'une variable dont on donne le nom en 1er paramètre (le 2ème paramètre indique une valeur par défaut)

Sérialisation spéciale – `Externalizable`

- **`readObject`** et **`WriteObject`** permettent de personnaliser la sérialisation de la classe elle-même ; la sérialisation des classes ancêtres est encore effectuée automatiquement
- Les classes qui implémentent l'interface **`Externalizable`** (hérite de **`Serializable`**) permettent d'avoir un processus de sérialisation totalement personnalisé

Sérialisation spéciale – `Externalizable`

- Cette interface comporte les 2 méthodes **`void readExternal(ObjectInput in)`** et **`void writeExternal(ObjectOutput out)`**
- **`ObjectInput`** et **`ObjectOutput`** sont 2 interfaces qui permettent de lire et d'écrire les types primitifs et les objets

Sérialisation spéciale – `Externalizable`

- Attention, **`readExternal`** et **`writeExternal`** sont publiques ! Il ne faut donc pas les utiliser pour des données sensibles
- Il ne faut pas oublier de prendre en charge la sérialisation des variables héritées des classes ancêtres

Vérification après désérialisation (1)

- Si on veut lancer une vérification automatique des instances d'une classe au moment de la désérialisation (après la construction de tout le graphe des objets), il faut que la classe implémente l'interface **`java.io.ObjectInputValidation`**
- Cette interface n'a qu'une seule méthode **`validateObject()`** qui doit renvoyer une **`InvalidObjectException`** s'il y a un problème, ce qui stoppera la désérialisation

Vérification après désérialisation (2)

- Le validateur est enregistré dans la méthode **`private readObject(ObjectInputStream)`**, étudiée précédemment, en appellant la méthode **`registerValidation(this, n)`**
- **`this`** est l'objet à valider et le 2ème paramètre ***`n`*** est un ordre de priorité qui fixe l'ordre d'exécution au cas où plusieurs objets auraient des validateurs (0 est standard) ; les plus grandes priorités sont exécutées en premier

Analyse lexicale

Introduction

- Cette section étudie des classes pour décomposer du texte en lexèmes :
  - classe **`StringTokenizer`**
  - classe **`StreamTokenizer`**
- Autres possibilités pour effectuer la même tâche (étudiées plus loin dans cette partie du cours) :
  - méthode **`split`** de la classe **`String`**
  - classe **`Scanner`**, introduite par le JDK 5

Séparateurs des données

- Rappel : les flots de caractères utilisent des séparateurs entre les données
- Les classes étudiées dans cette section facilitent l'extraction de données séparées par des séparateurs

Avertissement

- La classe **`StringTokenizer`** ne supporte pas l'utilisation d'expressions régulières et elle peut donc souvent être avantageusement remplacée par l'utilisation de méthode **`split`** de la classe **`String`** ou par la classe **`Scanner`**
- Cependant la connaissance de cette classe peut être utile pour comprendre du code déjà écrit

Analyse lexicale d'une chaîne

```
import java.util.StringTokenizer;
public class Decomposition {
  public static void main(String[] args) {
    String donnees =
      "Chiffres : 12,,689\n155";
    StringTokenizer st =
      new StringTokenizer(donnees, " ,\n");
    while (st.hasMoreTokens()) {
      String token = st.nextToken();
      System.out.println(token);
    }
  }
}
```

Pas **`java.io`**

Que sera-t-il affiché ?

```
Chiffres
:
12
689
155
```

2 séparateurs accolés sont considérés comme 1 seul séparateur

Analyse lexicale d'une chaîne

Que sera-t-il affiché ?

```
. . .
String donnees =
  "Chiffres : 12,,689\n155";
StringTokenizer st =
  new StringTokenizer(donnees, " ,\n",
                      true);
while (st.hasMoreTokens()) {
  String token = st.nextToken();
  System.out.println(token);
}
```

Les séparateurs sont des lexèmes (*tokens*)

```
Chiffres

:

12

,

,

689


155
```


Analyse lexicale d'un flot

- La classe **`java.io.StreamTokenizer`** possède un constructeur **`public StreamTokenizer(Reader r)`** qui permet de faire une analyse syntaxique du flot de caractères associé à **`r`**
- Cette classe offre plus de souplesse que **`StringTokenizer`** (surtout pour analyser des programmes Java ou C) mais est plus complexe ; elle ne sera pas étudiée dans ce cours

Entrées et sorties sur clavier - écran

Lecture caractère par caractère

```
int n; char car; String s = "";
try {
  while (true) {
    int n = System.in.read();
    if (n == -1) break;
    car = (char)n;
    s += car;
  }
}
catch(IOException e) {
  System.err.println("Erreur I/O" + e);
}
```

Cast pour avoir un **char**

Entrées-sorties sur clavier-écran

```
BufferedReader br =
  new BufferedReader(
    new InputStreamReader(System.in));
System.out.print("Entrez votre nom : ");
String nom = br.readLine();
System.out.print("Entrez votre âge : ");
String age = br.readLine();
System.out.println(nom + " a "
  + Integer.parseInt(age) + " ans.");
```


Entrées-sorties sur clavier-écran

printf étudié dans la partie 2 de ce support

```
Scanner sc = new Scanner(System.in);
System.out.printf("%-20s", "Votre nom : ");
String nom = sc.nextLine();
System.out.printf("%-20s", "Et votre âge :
  ");
int age = sc.nextInt();
System.out.printf("%s a %d ans", nom, age);
```

Depuis le JDK 5, le plus simple est d'utiliser la classe **Scanner** (étudiée dans la partie 2 de ce support) car on n'a pas besoin de gérer les **IOException**

Petite colle !

- Que signifie « **System.out.println()** » ?
- System est une classe du paquetage **java.lang**
- Elle est **final** et non instanciable. Elle comprend (entre autres) :
  - 3 variables de classe : **in**, **out**, **err** pour les voies standard d'entrée, sortie et de messages d'erreurs
- Déclaration de la variable **out** :
  public static PrintStream out;
- **println()** est une méthode de la classe **PrintStream**

A propos de **System.in**, **out** et **err**

- On peut rediriger les voies standard des entrées, sorties et erreurs par les méthodes **setIn(InputStream)**, **setOut(PrintStream)** et **setErr(PrintStream)** de la classe **System**

Console

- Depuis le JDK 6 la classe **`java.io.Console`** permet de lire et d'écrire des caractères sur la console (clavier – écran)
- En particulier elle permet de saisir un mot de passe tapé sur le clavier sans qu'il ne s'affiche sur l'écran

Exemple

```
Console console = System.console();
if (console == null)
  return;
console.printf("%s", "Mot de passe ? ");
char[] mdp = console.readPassword();
String mdps = new String(mdp);
console.printf("%s, %s", nom, mdps);
```


Nouveau paquetage **`java.nio`**

Présentation

- Ce paquetage reprend toute l'architecture des classes pour les entrées/sorties
- Il utilise la notion de canal (*channel*) et de buffer
- Il utilise les possibilités avancées du système d'exploitation hôte pour optimiser les entrées-sorties et offrir plus de fonctionnalités

Utilisation

- Il a été écrit pour
  - permettre des entrées-sorties non bloquantes
  - améliorer les performances, en particulier pour les serveurs fortement chargés
  - bloquer des portions de fichiers
- Il est un peu plus complexe à utiliser et pas nécessaire si ces fonctionnalités ne sont pas recherchées
- Il n'est pas étudié dans ce cours

Classe **`File`**

Avertissement

- Il vaut mieux utiliser la nouvelle API NIO.2 du JDK 7 (étudiée au début de ce support) que la classe **`File`**
- La méthode **`toPath()`** de la classe **`File`** renvoie une instance de **`Path`** ; elle peut être utile pour faire le pont avec NIO.2
- Cette section vous aidera à comprendre les nombreuses lignes de code écrites avec les versions précédentes du JDK

Classe **`File`**

- Cette classe représente la notion de fichier, indépendamment du système d'exploitation
- Un fichier est repéré par un chemin abstrait composé d'un préfixe optionnel (nom d'un disque par exemple) et de noms (noms des répertoires parents et du fichier lui-même)
- Attention,
  `File fichier = new File("/bidule/truc");`
  ne lève aucune exception si **`/bidule/truc`** n'existe pas dans le système de fichier

Constructeurs

- Les chemins passés en premier paramètre peuvent être des noms relatifs ou absolus
- **`File(String chemin)`**
- **`File(String cheminParent, String chemin)`**
- **`File(File parent, String chemin)`**

Portabilité pour les noms de fichiers

- La classe **`File`** offre des facilités pour la portabilité des noms de fichiers
- Le caractère séparateur pour le nom d'un fichier est donné par les constantes **`File.separator`** (de type **`String`**) ou **`File.separatorChar`** (de type **`char`**) (« **`/`** » pour Unix, « \ » pour Windows)
- Le caractère pour séparer des chemins (par exemple pour *classpath* ou *path*) est donné par **`File.pathSeparator`** et **`File.pathSeparatorChar`**

Noms relatifs

- Les noms relatifs sont relatifs au répertoire courant (propriété système **`user.dir`**)
- Le répertoire courant est généralement le répertoire dans lequel la JVM a été lancée (commande java), mais est difficile à maîtriser dans les environnements complexes d'exécution (Java EE par exemple)
- Un problème donc !

`new File(String)` est à éviter !

- Malgré les facilités pour la portabilité offertes par la classe **`File`**, l'utilisation de noms de fichiers « en dur » rend souvent un programme plus difficile à réutiliser
- En effet, donner des noms absolus, et même relatifs, nuit à la souplesse comme on le verra dans la section suivante « Noms de fichiers » de ce supports de cours

Fonctionnalités de la classe **`File`**

- Elle permet d'effectuer des manipulations sur les fichiers et répertoires considérés comme un tout (mais pas de lire ou d'écrire le contenu) :
  - lister un répertoire
  - supprimer, renommer un fichier
  - créer un répertoire
  - créer un fichier temporaire
  - connaître et positionner les droits que l'on a sur un fichier (lecture, écriture) (depuis JDK 6)
  - etc.

Méthodes informatives

- **`boolean exists()`**
- **`boolean isDirectory()`**
- **`boolean isFile()`**
- **`boolean canRead()`**
- **`boolean canWrite()`**
- **`boolean canExecute()`**
- **`long lastModified()`**
- **`long length()`**

Méthodes pour des actions

- **`boolean delete()`** (**`true`** si suppression réussie)
- **`boolean renameTo(File`** ***`nouveau`*****`)`** (**`true`** si suppression réussie)
- **`boolean mkdir()`**
- **`boolean mkdirs()`** (peut créer des répertoires intermédiaires)
- **`boolean setLastModified(long`** ***`temps`*****`)`**
- **`boolean setReadOnly()`**

Méthodes pour des actions

- **`static File createTempFile(String prefixe, String suffixe)`** crée un fichier dans le répertoire pour les fichiers temporaires, avec un nom unique
- **`void deleteOnExit()`** le fichier qui reçoit ce message sera supprimé à la fin de l'exécution

Méthodes pour les noms

- **`String getName()`** (nom terminal)
- **`String getPath()`** (chemin absolu ou relatif)
- **`File getAbsoluteFile()`**
- **`String getAbsolutePath()`**
- **`String getParent()`**
- **`File getParentFile()`**
- **`URL toURL()`** (de la forme **`file:`*`url`***)

Fichiers d'un répertoire

- Le résultat est **`null`** si **`this`** n'est pas un répertoire, et un tableau de taille 0 si le listing est vide
- **`String[] list()`** (noms terminaux)
- **`String[] list(FileNameFilter filtre)`** (seulement certains fichiers)
- **`File[] listFiles()`** (**`File`** avec des noms relatifs ou absolus, suivant **`this`**) ; variantes avec **`FileNameFilter`** et **`FileFilter`**

Modifier les permissions

- Depuis le JDK 6, il est possible de positionner les permissions des fichiers avec les méthodes **`setWritable`**, **`setReadable`**, **`setExecutable`**

Exemple d'utilisation de **`File`**

```
public boolean isLisible(String nomFichier) {
  File fichier = new File(nomFichier);
  if (!fichier.isFile())
    return false;
  return fichier.canRead();
}
```