

Interface avec le système, fichiers .jar, propriétés, ressources

Université de Nice - Sophia Antipolis

Version 2.2 – 24/11/07

Richard Grin

- Cette partie du cours décrit comment un programme Java peut s'insérer dans l'environnement dans lequel il tourne, et échanger des informations avec cet environnement

Plan de la section

- Classes **System** et **Runtime**
- Exécution d'un programme externe
- Fichiers JAR
- Propriétés (variables d'environnement)
- Interface avec les autres langages (C, C++, assembleur, etc.)

Classes System et Runtime

Classe `java.lang.System`

- Cette classe n'est jamais instanciée ; toutes les méthodes sont **public static**
- Cette classe fournit
 - la méthode **exit** pour arrêter l'exécution de la JVM
 - des références aux voies standard d'entrées-sorties
 - des méthodes pour lire et écrire des propriétés (étudiées plus loin dans cette section)
 - des méthodes diverses pour installer un gestionnaire de sécurité (voir cours sur la sécurité), charger des librairies, lire l'heure système, lancer le ramasse-miettes, copier des tableaux en blocs,...

Voies d'entrées-sorties standard

- 3 variables `public` correspondent aux 3 voies standards :

```
public static final PrintStream  
    in, out, err
```

- 3 méthodes `public static` permettent de rediriger ces voies :

```
- void setOut(PrintStream out)
```

```
- void setIn(InputStream in)
```

```
- void setErr(PrintStream err)
```

Quelques méthodes de `System`

- `void exit(int statut)` ; le statut est le code retour renvoyé par java (0 indique un déroulement normal)
- `native void arraycopy(Object src, int positionSrc, Object dst, int positionDst, int longueur)`
- `native long currentTimeMillis()`
- `void gc()`

Quelques méthodes de `System`

- `String getenv(String nomVariable)`
récupère la valeur d'une variable d'environnement du système d'exploitation ;
il vaut mieux utiliser l'option `-D` de java
(voir les propriétés plus loin dans ce cours)
- `Map<String, String> getenv()`
récupère les valeurs de toutes les variables d'environnement
- `void setSecurityManager(
SecurityManager s)`

Classe `java.lang.Runtime`

- Chaque programme Java est associé à une instance unique de la classe `Runtime`, obtenue avec la méthode
`public static Runtime getRuntime()`
- Cette instance donne au programme un accès au système d'exploitation dans lequel il s'exécute
- La classe `System` fait appel à cette instance pour exécuter plusieurs de ses méthodes

Quelques méthodes de `Runtime`

- `exec` permet de lancer un programme externe (étudié dans la section suivante)
- `long freeMemory()` retourne une approximation du nombre d'octets disponibles
- `long totalMemory()` retourne la mémoire totale occupée par la JVM
- `void gc()` demande à la JVM d'essayer de libérer de la place avec le ramasse-miette
- `void traceInstructions(boolean on)` demande à la JVM d'indiquer les instructions exécutées
- `void traceMethodCalls(boolean on)` demande à la JVM d'indiquer les appels de méthodes

Programmes externes

Exécuter un programme externe

- La classe `java.lang.Runtime` contient des méthodes `exec()` pour exécuter un programme externe à la JVM :
 - `Process exec(String commande)`
 - `Process exec(String[] commande)` : la commande sous la forme d'un tableau de chaînes
 - `Process exec(String commande, String[] env)` et `Process exec(String[] commande, String[] env)` : on peut passer des variables d'environnements sous la forme `var=valeur`

Classe `java.lang.Process`

- Les méthodes `exec` de `Runtime` renvoient un objet de type `Process`
- Les méthodes de la classe `Process` offrent une interface minimum avec le processus lancé :
 - `exitValue()` récupère le code retour du processus (erreur si le processus n'est pas terminé)
 - `waitFor()` attend la fin du processus et récupère le code retour
 - `destroy()` supprime le processus

Classe `java.lang.Process`

- On peut aussi agir sur les 3 voies standard du processus :
 - `get{Input|Output|Error}Stream()` récupère les 3 voies standard du processus (sortie, entrée et erreur)
 - Attention ! `getInputStream` se branche sur la **sortie** standard du processus (on se place du point de vue du programme Java)
 - Il est conseillé de les récupérer rapidement ces voies pour éviter le blocage de certains processus

Exemple de processus externe

```
Process p = Runtime.getRuntime().exec("who");
BufferedReader br = new BufferedReader(
    new InputStreamReader(p.getInputStream()));
ArrayList utilisateurs = new ArrayList();
String utilisateur;
while ((utilisateur = br.readLine()) != null) {
    utilisateurs.add(utilisateur);
    System.out.println(utilisateur);
}
```

Attention, risque de
blocage de **readLine**
avec **command.com** du DOS

Échanges avec le programme externe

- Si on a lancé un programme qui accepte des entrées par la voie standard d'entrée (un *shell* Unix par exemple), on peut lui envoyer des messages en utilisant `getOutputStream` :

```
PrintWriter pw = new PrintWriter(  
    new OutputStreamWriter(  
        p.getOutputStream()));  
pw.println(message);  
pw.flush();  
...
```

ProcessBuilder

- Elle a été introduite par le JDK 5 pour offrir plus de possibilités que celles offertes par la méthode **exec** de **Runtime**
- Sa méthode **start ()** renvoie une instance de **Process**

ProcessBuilder

- On peut
 - modifier le répertoire courant,
 - rediriger la voie standard des erreurs vers la voie de sortie standard,
 - récupérer l'environnement du processus (les valeurs des variables) et le modifier (par l'intermédiaire d'une Map)
- Cette classe n'est pas protégée contre les accès multiples (par plusieurs threads)

Exemple de base

```
Process p =
    new ProcessBuilder("ipconfig", "/all")
        .start();
InputStream is = p.getInputStream();
BufferedReader br = new BufferedReader(
    new InputStreamReader(is));
String line;
while ((line = br.readLine()) != null) {
    System.out.println(line);
}
```

Exemple avec modification de l'environnement

```
ProcessBuilder pb =
    new ProcessBuilder("uneCommande",
                       "arg1", "arg2");
// Modification de l'environnement
Map<String, String> env = pb.environment();
env.put("VAR1", "myValue");
env.remove("VAR2");
env.put("VAR2", env.get("VAR1") + "suffix");
pb.directory("unRep"); // répertoire courant
Process p = pb.start();
```

Classe Desktop

- Introduite avec le JDK 6, elle permet d'ouvrir un fichier avec l'application qui lui est liée dans le système, par exemple, d'ouvrir un fichier « .doc » avec Open Office ou Word
- Une fois l'application lancée, le code Java ne peut plus communiquer avec elle

Commandes du shell

- Si on veut lancer une commande du langage de commande du shell (par exemple `cd`), il faut lancer l'exécution du shell avec la commande en argument
- Exemple sous Windows :

```
Runtime runtime =  
    Runtime.getRuntime();  
runtime.exec(new String[] {  
    "cmd.exe", "/C", "dir" });
```

Fichiers .jar

Fichiers `.jar`

- Les fichiers `.jar` (*Java ARchive*) sont utilisés pour conserver **en un seul fichier**, plusieurs fichiers utilisés par des programmes Java :
 - fichiers `.class` Java,
 - images,
 - son,
 - ...

Format des fichiers `.jar`

- « Repliage » de répertoires et compression des fichiers au format *zip*
- Ce fichier zip contient un fichier particulier à l'emplacement (attention, la casse des lettres peut varier !)

META-INF/MANIFEST.MF

- Ce fichier **MANIFEST.MF** contient des informations sur le fichier jar et les fichiers contenus dans le fichier jar

Avantages des fichiers `.jar` (1)

- Rapidité de chargement des fichiers, spécialement si on utilise le protocole HTTP (avec des applets)
- Compression des données (moins de place sur disque)
- Format standard de distribution des applications Java (donc portabilité)

Avantages des fichiers `.jar` (2)

- Grâce au fichier MANIFEST.MF, on peut
 - *signer* des fichiers (par exemple, pour avoir des *applets* sécurisées)
 - gérer des versions
 - donner des informations utiles pour exécuter l'application (classe principale, *classpath*)
 - protéger des paquetages contre l'ajout de classes

Utilitaire `jar` pour fichiers `.jar`

- La syntaxe ressemble à la commande `tar` d'Unix
- 3 modes de travail :
 - création (option `c`)
 - extraction (option `x`)
 - listing (option `t`)
 - modification (option `u`)

Utilitaire `jar`

```
jar {c|t|x|u} [f fichier-archive.jar] [fichiers...]
```

- **c** pour créer un fichier archive à partir des *fichiers*
 - si un des *fichiers* est un répertoire, toute l'arborescence est archivée (désarchivée pour les options **x** ou **t**)
- **x** (resp. **t**) pour extraire (resp. lister) tous les fichiers de l'archive *fichier-archive.jar*
 - si on donne une liste de *fichiers*, seuls ces *fichiers* de l'archive sont extraits (resp. listés)
- **f** *fichier-archive.jar* indique avec quel fichier archive travailler
 - par défaut, *jar* travaille avec l'entrée ou la sortie standard

Utilitaires pour fichiers `.jar` (2)

- `-u` (*update*) pour modifier un fichier archive à partir des *fichiers*. On peut ainsi ajouter des fichiers ou remplacer des fichiers par une nouvelle version
- `-m fichier-manifest` permet de d'indiquer un fichier qui sera le fichier **MANIFEST.MF** du fichier jar
- `-v` (*verbose*) donne plus d'information sur l'opération pendant son exécution, ou sur le listing

Noms des entrées du jar

- Correspondent au chemin donné pour désigner ce qu'il faut mettre dans le jar
- Si on tape par exemple
`jar uvf truc.jar rep/fichier truc`
on aura 2 entrées `rep/fichier` et `truc`
- Si on veut plutôt une entrée `fichier`, il faut utiliser l'option `-C` qui fait un changement temporaire de répertoire (par rapport au répertoire courant d'exécution) pendant la compression :
`jar uvf truc.jar -C rep fichier`

Les noms des entrées sont relatifs

- Les noms des entrées d'un jar sont relatifs et pas absolus
- Si on tape par exemple
`jar uvf truc.jar /usr/bin/ls`
on aura une entrée `usr/bin/ls` dans le jar
- Sous Windows, on peut cependant avoir des entrées du type
`C:/rep/machin`

Exemple de création de fichier *jar*

- Création d'un fichier **f1.jar** contenant tous les fichiers placés dans l'arborescence du répertoire **classes** et le fichier **MANIFEST.MF** placé sous le répertoire **src** :

```
jar cvfm f1.jar src\MANIFEST.MF -C classes .
```

- Remarquez l'option **-C** qui permet de ne pas placer le répertoire **classes** lui-même dans le fichier **jar**

Autre exemple de création

- Création d'une archive **f1.jar** (**v** est l'option « verbeuse ») contenant tous les fichiers **.class** du répertoire courant et tous les fichiers de l'arborescence du répertoire **images**

```
jar cvf f1.jar *.class images
```

Exemples de modification

- Ajout du fichier `truc.class`

```
jar uvf f1.jar truc.class
```

- Remplacement du fichier `truc.class` de l'archive par une nouvelle version

```
jar uvf f1.jar truc.class
```

Exemples d'extraction de fichiers depuis un fichier *jar*

- Lister les fichiers de l'archive **f1.jar**

```
jar tvf f1.jar
```

- Extraire tous les fichiers de l'archive (attention, cette commande écrase tous les fichiers éventuels de même nom qu'un fichier extrait)

```
jar xvf f1.jar
```

- Extraire certains fichiers de l'archive

```
jar xvf f1.jar META-INF/MANIFEST.MF
```

```
jar xvf f1.jar Truc.java images/im1.gif
```

Modifier le fichier **MANIFEST**

- Modifier le fichier **MANIFEST.MF** avec le fichier `rep/manifest` ; dans l'exemple suivant, **m** étant placé avant **f** dans la commande, on place le fichier « manifest » avant le fichier jar

```
jar umvf rep/manifest f1.jar
```

- Autre commande équivalente (**f** et **m** dans un autre ordre) :

```
jar ufmv f1.jar rep/manifest
```

Fichier **MANIFEST** par défaut

- Une entrée **META-INF/MANIFEST.MF** est créée par défaut si on ne donne pas d'option **m** lors de la création du fichier *jar*
- Ce fichier **MANIFEST.MF** ne contient aucune information spéciale sur le fichier jar, si ce n'est la version du format jar et celle du java qui l'a créé :

```
Manifest-Version: 1.0
```

```
Created-by: 1.4.1 (Sun Microsystems Inc.)
```

Fichier **MANIFEST**

- Un fichier **MANIFEST.MF** peut contenir les informations suivantes :
 - version du format *jar*
 - classe principale d'une application, répertoires liés à une *applet*, etc.
 - *checksum* pour vérifier les fichiers contenus dans le fichier *jar* (depuis la version Java 2, ce *checksum* est dans le fichier *manifest* seulement pour les fichiers *jar* signés)

Pour pouvoir disposer des classes d'un fichier jar

- Dans le cas d'une *application*, il suffit d'ajouter le fichier jar dans le *classpath*
- On utilise pour cela les options **-cp** ou **-classpath** de la commande *java* (on peut aussi utiliser la variable **CLASSPATH**)
- Plus rarement, on place le fichier jar dans le répertoire des extensions
- Voir aussi plus loin la notion de fichier jar d'extension (quand l'application est dans un jar)

Exécution d'un fichier *jar*

- La commande *java* a une option **-jar** qui permet de donner un nom de fichier **.jar** au lieu du nom d'une classe comme classe initiale d'exécution
- Le fichier « *manifest* » du fichier **.jar** doit contenir une en-tête **Main-Class** qui indique la classe qui devra être exécutée avec l'option **-jar**. Par exemple :

```
Main-Class: fr.unice.TestClass
```

Pas d'espace avant le « : » !

Problèmes avec le fichier MANIFEST

- Pour éviter des problèmes fréquemment rencontrés :
 - **pas d'espace avant les « : » des entrées du jar** (**Main-Class:** par exemple)
 - terminer la dernière ligne du fichier par un passage à la ligne
 - attention aux majuscules/minuscules dans les noms de fichiers et de répertoires

Jar exécutable sous Windows

- Sous Windows les fichiers Jar sont associés à l'exécutable `javaw -jar` lors de l'installation de Java
- Si le fichier Jar a une entrée `Main-Class:`, il peut donc être lancé en faisant un double clic sur son icône

Classpath avec l'option **-jar**

- Quand on lance une application avec l'option **-jar**, le *classpath* se limite au « répertoire » racine placé dans le fichier jar (sauf s'il y a une entrée **Class-Path** dans le jar ; voir transparents suivants)
- Même une option **-classpath** sera ignorée !
- Sun étudie actuellement le changement de cette fonctionnalité (le jar indiqué par l'option **-jar** au *classpath* serait simplement ajouté au *classpath*) ; toujours pas fait dans la version 1.5...

Fichier *jar* d'extension (1)

- Le fichier *manifest* du fichier JAR peut contenir une entrée **Class-Path** qui indique les **URL** d'autres fichiers *jar* (ou zip) ou de répertoires (suffixes « / » obligatoire) qui pourront être utilisés pour trouver les classes
- Les classes et ressources seront donc recherchés dans le jar et aussi dans les endroits indiqués par les entrées de **Class-Path**

Fichier *jar* d'extension (1)

- Les fichiers doivent être dans le système de fichiers **local**
- Les URL relatives doivent être relatives au répertoire du fichier jar qui les contient
- Attention, il s'agit d'URL donc le séparateur de noms de fichiers est « / », même sous Windows et les noms absolus Windows sont de la forme **/C:/rep1...**

Fichier *jar* d'extension (2)

- Si c'est le jar d'une applet, le navigateur devra les charger en même temps que l'archive de l'*applet*
- Un fichier manifest peut contenir plusieurs entrées **Class-Path**: séparées par un ou plusieurs espaces
- Par exemple :

```
Class-Path: rep/ rep/truc2.jar
```

Dépendance avec d'autres jars

- Mettre un jar dans un jar ne sert à rien ; ça ne marche pas !
- Si les classes d'un jar dépendent de bibliothèques contenues dans des jars, 2 solutions :
 - utiliser une entrée **Class-Path** dans le manifest (solution la plus utilisée)
 - décompacter les bibliothèques et les recomparer dans un seul jar, avec le jar de l'application principale

Dépendance avec d'autres jars

- La 1ère solution (**Class-Path**) est la plus souple car elle permet d'utiliser facilement les nouvelles versions des librairies utilisées
- Mais cette solution a l'inconvénient de ses avantages car les nouvelles versions des librairies peuvent créer des incompatibilités avec l'application principale
- La 2ème solution fige les versions des librairies, ce qui a ses avantages et ses inconvénients

Information sur le contenu

- Le fichier manifest d'un jar peut contenir des informations sur le contenu du jar
- On peut ainsi indiquer un numéro de version pour la spécification ou l'implémentation

Exemple

Manifest-Version: 1.0

Created-By: 1.5.0_01 (Sun Microsystems)

Name: java/util/

Specification-Title: Utility Classes

Specification-Version: 1.2

Specification-Vendor: Sun Microsystems

Implementation-Title: java.util

Implementation-Version: build57

Implementation-Vendor: Sun Microsystems

Signature d'un fichier jar

- En Java 2, l'outil *jarsigner* permet de signer les fichiers jar (voir cours sur la sécurité)

Exécution d'une *applet* contenue dans un fichier *jar*

```
<applet  
  code="fr.unice.p1.UnecLasse.class"  
  archive="jar/fichier.jar"  
  width=120 height=120>  
</applet>
```

Ressources placées dans un fichier jar

- On peut situer une ressource placée dans un fichier jar avec la méthode « **URL getResource(String)** » de la classe **Class**
- On peut alors utiliser la ressource après avoir récupéré son URL
- Cette méthode fonctionne avec les applets, les applications, qu'elles soient placées dans un fichier jar ou non
- Plus de précisions dans la section « ressources » de la fin de ce support

Jar et images

- Le cours sur les interfaces graphiques indique comment utiliser les images placées dans un fichier jar (on utilise le chargeur de classe pour la recherche de ressources) :

```
URL url =  
    getClass().getResource(nomFichier);  
ImageIcon icone = new ImageIcon(url);
```

- On peut ainsi, par exemple, charger les icônes utilisées par les barres d'outils

Code pour lire un fichier jar (1)

```
import java.net.JarURLConnection;  
import java.util.zip.ZipEntry;  
import java.util.jar.JarFile;  
  
. . .  
URL url =  
    new URL("jar:file:/chemin/f.jar!/");  
JarURLConnection jarConnection =  
    (JarURLConnection)url.openConnection();  
JarFile jf = conn.getJarFile();  
// ou jf = new JarFile("truc.zip");
```

Code pour lire un fichier jar (2)

```
// Lecture séquentielle des entrées :
Enumeration e = jf.entries();
while (e.hasMoreElements()) {
    JarEntry entree =
        (JarEntry)e.nextElement();
    String nomEntree = entree.getName();
    String nom = entry.getName();
    // Lecture du contenu d'une entrée
    InputStream is =
        jf.getInputStream(entry);
    . . .
}
```

Code pour lire un fichier jar (3)

```
// Ou accès direct à une entrée :  
JarEntry entry =  
    jf.getJarEntry("rep\\f1.txt");  
InputStream is =  
    jf.getInputStream(entry);  
// On lit le contenu de l'entrée  
• • •
```

Récupérer les informations du jar

- La classe `java.lang.Package` permet de récupérer les informations du jar :

```
Package pkg =  
    Package.getPackage("fr.truc");  
String nom =  
    pkg.getSpecificationVersion();
```

Classe `JarURLConnection`

- Classe de base abstraite pour représenter une connexion à un jar ou une entrée d'un jar
- `url.openConnection()` (classe `URL`) renvoie une instance de ce type si l'URL est celui d'un jar
- Elle contient des méthodes qui facilitent le travail avec un jar, par exemple, pour accéder aux attributs du fichier **MANIFEST**
- Elle se trouve dans le paquetage `java.net`

Format d'URL pour une entrée de jar

- `jar:<url du jar>!/{entrée du jar}`
- Par exemple,
`jar:http://truc.fr/rep/f.jar!/fr/truc/Classe.class`
- Un fichier jar :
`jar:http://truc.fr/rep/f.jar! /`
- Un répertoire dans un fichier jar (se termine par un « / ») :
`jar:http://truc.fr/rep/f.jar! /fr/truc/`
- Un répertoire local (sur la même machine)
`jar:file:rep1/rep2/`

Propriétés

Propriété

- Un programme Java s'exécute dans un environnement connu par le programme sous la forme de propriétés
- Une propriété est caractérisé par
 - un nom
 - une valeur
- Le nom d'une propriété est souvent hiérarchique. Par exemple, **java.class.version**

Types de propriétés

- Les propriétés système décrivent l'environnement du système dans lequel s'exécute le programme Java ; exemple : **`user.name`**, **`line.separator`**, **`file.separator`**, etc.
- Le programme (ou l'utilisateur) peut avoir ses propres propriétés qui ne sont utilisés que par lui-même ; exemple : **`editeur.son`** (pour indiquer si le programme émettra des sons ou non)

Les propriétés

- La classe `java.util.Properties` (classe fille de `Hashtable`) a été conçue pour
 - manipuler des ensembles de propriétés
 - les conserver entre 2 invocations d'un programme, en les enregistrant dans des fichiers

Propriétés standard du système

- La JVM récupère automatiquement des attributs standard du système :

<code>java.version</code>	<code>java.class.version</code>	
<code>java.vendor</code>	<code>java.vendor.url</code>	
<code>java.home</code>	<code>java.class.path</code>	
<code>os.name</code>	<code>os.arch</code>	<code>os.version</code>
<code>file.separator</code>	<code>line.separator</code>	<code>path.separator</code>
<code>user.name</code>	<code>user.home</code>	<code>user.dir</code>
<code>user.language</code>	<code>user.region</code>	<code>user.timezone</code>
<code>file.encoding</code>	<code>file.encoding.pkg</code>	

- A utiliser pour la portabilité des applications :
`file.separator`, `line.separator` (fins de ligne)
- `user.dir` est le répertoire courant

Récupérer les propriétés standard du système

- On utilise 2 méthodes **static** de la classe **System** :
- **String getProperty(String nomPropriete)** renvoie la valeur d'une propriété système sous forme de **String**
- **Properties getProperty()** renvoie toutes les propriétés système

Valeurs de propriétés données sur la ligne de commande

- On peut donner d'autres valeurs de propriétés système au lancement de l'interprète Java, avec l'option **-D** de **java** :

```
java -Dediteur.prop1=25 -Dediteur.son=off Edit
```

Format des fichiers de propriétés (1)

- Un programme Java peut aussi lire des valeurs de propriétés depuis des fichiers de propriétés :

```
# Propriétés pour l'éditeur Java
editeur.son = off
# La valeur peut contenir des espaces
editeur.prop1 = mot1 mot2
...
```

- Le séparateur entre le nom et la valeur peut être « = », « : » ou tabulation
- Les lignes de commentaires commencent par # ou !

Format des fichiers de propriétés (2)

- Une ligne par propriété ; elle peut se continuer à la ligne suivante si elle se termine par \
- Il faut placer un \ avant les caractères #, !, = et : s'ils n'ont pas leur signification habituelle
- Un simple \ doit être doublé : « \\ »
- Par exemple, pour désigner un fichier « C:\rep1\fichier » sous Windows :
fichier = C:\\rep1\\fichier

Format des fichiers de propriétés (3)

- Le flot est écrit avec le codage 8859-1
- Attention, les espaces de fin de ligne risquent d'être incluses dans les valeurs des propriétés

Exemple d'utilisation des propriétés

```
// Crée une instance de Properties qui sera utilisé pour
// modifier les propriétés système, avec les
// propriétés système actuelles comme valeur par défaut
Properties props =
    new Properties(System.getProperties());
// Charge des nouvelles propriétés
BufferedInputStream bis =
    new BufferedInputStream(
        new FileInputStream("f.properties"));
props.load(bis);
bis.close();
// Positionne la combinaison des propriétés comme
// nouvelles propriétés système
System.setProperties(props);
```

Exemple d'utilisation des propriétés

```
// Lit la valeur d'une propriété
String son = props.getProperty("editeur.son");
// Modifie la propriété
props.setProperty("editeur.son", "off");
// Sauvegarde les nouvelles propriétés
BufferedOutputStream bos =
    new BufferedOutputStream(
        new FileOutputStream("f.properties"));
props.store(bos, "Propriétés système avec son");
bos.close();
```

Mis en commentaire (#)
en 1ère ligne du fichier

Propriétés et ressources

- Travailler avec un nom de fichier pour conserver des propriétés n'est pas souple (voir cours sur les fichiers)
- Il vaut mieux utiliser des noms de ressources (voir section suivante)
- Le transparent suivant récrit l'exemple précédent pour la lecture des propriétés dans un fichier ; ce fichier est maintenant désigné par un nom de ressources et plus par un nom relatif

Exemple de chargement de propriétés en utilisant un nom de ressources

```
BufferedInputStream bis =  
    new BufferedInputStream(  
        getClass().getResourceAsStream  
            ("f.properties"));  
props.load(bis);  
bis.close();
```

Le fichier de propriétés est
situé dans le même
répertoire que la classe qui
contient ce code

Énumérer des propriétés

- La méthode `propertyNames()` de la classe `Properties` renvoie une `Enumeration` pour énumérer tous les noms des propriétés d'une instance de `Properties`
- On peut aussi afficher toutes les propriétés sur un `PrintStream` ou un `PrintWriter` avec les méthodes `list(PrintStream ps)` ou `list(PrintWriter pw)` ; par exemple, `props.list(System.out)`

Propriétés d'un autre type que `String`

- `System.getProperty` permet de lire la valeur d'une propriété sous forme d'une chaîne de caractères
- La classe `java.awt.Font` contient la méthode `static Font getFont(String nomPropriété)` qui renvoie la fonte spécifiée par la valeur de la propriété `nomPropriété` (nom d'une fonte)
- D'autres méthodes permettent de lire des valeurs de propriétés de type `Integer`, `Boolean`, `java.awt.Color` ou autres

Fichiers de propriétés XML

- Les méthodes **storeToXML** et **loadFromXML** permettent de conserver les propriétés dans des fichiers XML

- Les fichiers qui contiennent des valeurs de propriétés sont un cas particulier de fichiers de ressources

Ressources

Définition

- Une **ressource** est un élément extérieur à un programme Java, utilisé par ce programme
- Par exemple, une classe Java peut utiliser un fichier de propriété, une image, un fichier qui contient du son, ou un fichier binaire quelconque
- Il est souvent intéressant de regrouper dans un fichier JAR les classes et les ressources utilisées par ces classes

Emplacement de la ressource

- Une ressource est associée à une classe (une des classes qui l'utilisera)
- Son emplacement est donc le plus souvent relatif à l'emplacement de cette classe :
 - dans le système de fichier local
 - dans un fichier JAR
 - sur un serveur Web

Méthode `getResource()`

- La classe **Class** contient une méthode qui permet d'obtenir une ressource en donnant un chemin « absolu », ou relativement à l'emplacement de la classe qui effectue le chargement :

URL `getResource(String nom)`

- Cette méthode très utile est souvent mal utilisée ; étudions-la en détails

Recherche de la ressource par la classe

- La classe va déléguer la recherche à son chargeur de classe
- Avant de déléguer, elle va faciliter la recherche de ressources dont l'emplacement est relatif à l'emplacement de son fichier **.class**

Préliminaire

- En fait, la recherche dépend du chargeur de la classe ; elle s'effectue le plus souvent dans le *classpath* mais elle peut aussi s'effectuer dans une base de données ou sur le réseau
- On va décrire d'abord la recherche effectuée par les chargeurs de classes habituels (en général des `UrlClassLoaders`), qui recherchent dans le *classpath*
- On verra ensuite le cas général pour un chargeur de classe quelconque

Nom d'une ressource

- Un nom **absolu** commence par un « / » :

`/images/truc.jpg`

dans ce cas, le nom indiqué désigne un chemin par rapport à un des répertoires indiqués dans le *classpath*

- Si le nom est relatif :

`images/truc.jpg`

le chemin est **relatif** au répertoire du paquetage de la classe qui charge la ressource :

- Attention, le séparateur est toujours « / », quel que soit le système d'exploitation

Pratiquement

- Lorsque l'on distribue une application (dans un jar ou non), on peut choisir 2 politiques pour les ressources :
 - placer toutes les ressources dans un répertoire à part (souvent nommé *resources*, ou *resources* en anglais) ; en ce cas, on utilisera des noms absolus
 - placer les ressources sous le même répertoire que les classes qui les utilise (souvent dans un sous-répertoire) ; on utilisera alors des noms relatifs

Dans un jar

Une arborescence dans un jar qui contient 2 répertoires p1 et images :

- p1

- p2

- C1.class
 - images

- i1.gif

- images

- i2.gif

- La classe C1 récupérera l'URL de `i1.gif` par :

```
getClass().getResource("images/i1.gif");
```

- et de `i2.gif` par :

```
getClass().getResource("/images/i2.gif");
```

(le / du début fait la différence)

Ressources dans un jar à part

- Dans le cas où on ne place pas l'application dans un jar, on peut placer les ressources dans un fichier jar ou zip
- Pour l'exécution il faudra alors placer le fichier jar ou zip dans le *classpath* et désigner les ressources par des noms absolus
- Si l'application est dans un jar, on peut faire de même en ajoutant dans le fichier Manifest du jar, le nom du fichier qui contient les ressources (entrée Class-Path)

Recherche d'une ressource

1. Obtenir l'objet **Class** de la classe qui charge la ressource

– si c'est la classe courante :

```
Class c = getClass();
```

– si c'est une autre classe :

```
Class c = NomClasse.class; // ou  
Class c = Class.forName("nomClasse");
```

2. Obtenir l'URL du fichier de ressource :

```
URL urlRess = c.getResource(nomRess);
```

Remarque

- Comme la recherche est déléguée au chargeur de classe de la classe, on pourrait penser utiliser explicitement ce chargeur :
`getClass().getClassLoader().getResource(...)`
- Mais si on fait ainsi, on perd la souplesse de l'utilisation des noms relatifs pour donner le chemin des ressources par rapport à l'emplacement de la classe
- De plus, on a besoin d'une permission spéciale (pour obtenir le chargeur de classes ; voir cours sur la sécurité)

Lire une ressource

- Quand on a l'URL, on peut lire le contenu de la ressource par

```
InputStream in = urlRess.openStream();
```

- La classe **Class** fournit aussi un raccourci qui renvoie directement un **InputStream** à partir d'un nom de ressource ; si **c** est la classe qui utilise la ressource :

```
InputStream in =  
    c.getResourceAsStream(nomRess);
```

Récupérer une ressource multimédia

- Les ressources multimédia peuvent se récupérer directement à partir de l'URL du fichier :
 - `getImage(urlRess)` de la classe Toolkit pour les images
 - `new ImageIcon(urlRess)` pour une icône
 - `getAudioClip(urlRess)` pour un fichier son (seulement dans une *applet* ; voir `javax.sound.sampled.AudioSystem` sinon)

Chargeur de classes quelconque

- Le nom passé à **getResource** peut être
 - **absolu** (commence par un « / ») : la recherche est effectuée par le chargeur de classes avec l'URL inchangé (le mode de recherche dépend du chargeur)
 - **relatif** (ne commence pas par un « / ») : le chemin associé au paquetage de la classe est ajouté au début ; par exemple `/fr/unice/toto/p1/` si la classe est du paquetage `fr.unice.toto.p1`, et ensuite la recherche se fait comme avec un chemin absolu

Noms de ressources sous Windows

- Un nom relatif : **rep1/rep2/...**
- Un nom absolu : **/C:/rep1/...**

URL d'une ressource d'un jar

! / sépare l'URL et l'entrée dans l'URL

- Format :

`jar:url!/[entrée]`

- Par exemple :

`jar:http://www.unice.fr/~toto/f.jar!
/fr/unice/fr/toto/Truc.class`

ou (désigne tout le fichier jar)

`jar:http://www.unice.fr/~toto/f.jar!`

- `jar:file:/rep1/rep2/f.jar`

Interface avec un autre langage que Java

- Nous n'étudierons pas cette fonctionnalité du langage Java
- *Java Native Interface* (JNI) du JDK fournit une API pour appeler dans un programme Java des méthodes écrites dans un autre langage (C le plus souvent)