

*Programmation JAVA avancée
Sun service formation*

Projet :
Programmation
JAVA avancée

Copyright
Sun Service Forma-
tion

Réf. Sun :
SL275

Révision : F-beta

Date : 18/2/99

*Sun
Microsystems*

*Programmation JAVA avancée
Sun service formation*

Projet :
Programmation
JAVA avancée

Copyright
Sun Service Forma-
tion

Réf. Sun :
SL275

Révision : F-beta

Date : 18/2/99

*Sun
Microsystems*

*Programmation JAVA avancée
Sun service formation*

Projet :
Programmation
JAVA avancée

Copyright
Sun Service Forma-
tion

Réf. Sun :
SL275

Révision : F-beta

Date : 18/2/99

*Sun
Microsystems*

*Programmation JAVA avancée
Sun service formation*

Projet :
Programmation
JAVA avancée

Copyright
Sun Service Forma-
tion

Réf. Sun :
SL275

Révision : F-beta

Date : 18/2/99

*Sun
Microsystems*

*Programmation JAVA avancée
Sun service formation*

Projet :
Programmation
JAVA avancée

Copyright
Sun Service Forma-
tion

Réf. Sun :
SL275

Révision : F-beta

Date : 18/2/99

*Sun
Microsystems*



Programmation JAVA avancée

Sun service formation

Projet : Programmation JAVA avancée
Copyright Sun Service Formation
Réf. Sun : SL275

Révision : F-beta
Date : 18/2/99

Sun Microsystems France



Programmation JAVA avancée

Sun service formation

Projet : Programmation JAVA avancée
Copyright Sun Service Formation
Réf. Sun : SL275

Révision : F-beta
Date : 18/2/99

Sun Microsystems France



Programmation JAVA avancée

Sun service formation

Projet : Programmation JAVA avancée
Copyright Sun Service Formation
Réf. Sun : SL275

Révision : F-beta
Date : 18/2/99

Sun Microsystems France



Programmation JAVA avancée

Sun service formation



Sun Microsystems France S.A.
Service Formation
143 bis, avenue de Verdun
92442 ISSY LES MOULINEAUX Cedex
Tel 01 41 33 17 17
Fax 01 41 33 17 20

Intitulé Cours : Programmation JAVA avancée
Copyright : Sun Service Formation
Réf. Sun : SL275

Révision : F-beta
Date : 18/2/99

Sun Microsystems France





Protections Juridiques

© 1998 Sun Microsystems, Inc.
2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

AVERTISSEMENT

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou de sa documentation associée ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a.

Des parties de ce produit pourront être dérivées du système UNIX® licencié par Novell, Inc. et du système Berkeley 4.3 BSD licencié par l'Université de Californie. UNIX est une marque enregistrée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company Ltd. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Sun, Sun Microsystems, le logo Sun, sont des marques déposées ou enregistrées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC, utilisées sous licence, sont des marques déposées ou enregistrées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

Les interfaces d'utilisation graphique OPEN LOOK® et Sun™ ont été développées par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant aussi les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui en outre se conforment aux licences écrites de Sun.

Le système X Window est un produit de X Consortium, Inc.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" SANS GARANTIE D'AUCUNE SORTE, NI EXPRESSE NI IMPLICITE, Y COMPRIS, ET SANS QUE CETTE LISTE NE SOIT LIMITATIVE, DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DES PRODUITS A RÉPONDRE A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ILS NE SOIENT PAS CONTREFAISANTS DE PRODUITS DE TIERS.



Applets (rappels), archives jar	12
Applets : lancement	13
Applets: protocoles	15
Applets : cycle de vie	17
Applets : demandes au navigateur.....	18
Applets : demandes du système de fenêtrage	19
Applets : exemple.....	20
Applets : utilisation d'archives Jar	21
Applets : Ressources dans une archive Jar	22
Le système de sécurité de Java.....	23
Applications autonomes: sécurité	24
Applications autonomes : archives Jar.....	25
Le package AWT (rappels), LayoutManagers	28
Accès aux manipulations graphiques	29
Les gestionnaires de Disposition (LayoutManager)	31
FlowLayout :	32
BorderLayout :	33
GridLayout :	34
CardLayout :	35
GridBagLayout :	36
Le traitement des événements AWT (rappels).....	40
Les événements	41
Modèle d'événements JDK 1.1	42
Comportement de l'interface graphique utilisateur Java.....	46
Tableau des interfaces de veille	47
Evénements générés par les composants AWT.....	48
Détails sur les mécanismes	49
Adaptateurs d'événements.....	50
Les composants SWING.....	54
Java Foundation Classes	55
Les Composants SWING	57
SWING : hiérarchie des composants.....	58
Une application Swing de base :	59
La classe JComponent	62
Les Threads.....	64
Concept de thread.....	65
Création d'un Thread Java.....	67
Demarrage d'un Thread Java	69
Contrôle de base des threads	72
D'autres façons de créer des threads.....	73
Etats d'un Thread (résumé).....	74
Accès concurrents	76



Utilisation du mot-clé synchronized	77
Interaction de threads- wait() et notify()	87
Pour assembler le tout.....	92
Classe SyncStack	98
Etats d'un Thread (résumé).....	101
Principes des Entrées/Sorties	104
Fots E/S avec Java.....	105
Streams de base	109
Flots d'entrée sur URL	111
Readers et Writers.....	112
Fichiers.....	114
Tests de fichiers et utilitaires	115
Fichiers à accès direct	116
La programmation réseau.....	120
Modèles de connexions réseau en Java.....	121
Programmation réseau en Java	122
Le modèle réseau de Java.....	123
Principe d'un Serveur TCP/IP.....	124
Principe d'un Client TCP/IP.....	125
échanges UDP.....	126
Exemple de Serveur UDP	127
Exemple de client UDP	129
UDP en diffusion (Multicast)	131
Exemple de Serveur Multicast	132
Exemple de client Multicast	133
Linéarisation des objets (Serialization).....	136
Introduction	137
Architecture de sérialisation.....	138
Ecriture et lecture d'un flot d'objets.....	142
Effets de la linéarisation.....	144
Personnalisation de la linéarisation.....	145
RMI (introduction technique)	148
Fonction de l'architecture RMI en Java.....	150
Packages et hiérarchies RMI.....	151
Création d'une application RMI	154
Création d'une application RMI	155
Sécurité RMI.....	175
JDBC (introduction technique)	178
Introduction	179
"Pilote" JDBC.....	180
Organigramme JDBC.....	182
Organigramme JDBC.....	183

Exemple JDBC	184
Création de pilotes JDBC	186
Pilotes JDBC	188
Instructions JDBC.....	191
Méthodes setXXX	193
Méthodes getXXX	198
Correspondance des types de données SQL en Java	199
Utilisation de l'API JDBC.....	200
Annexe : JNI.....	202
Pourquoi réaliser du code natif?.....	203
un exemple : "Hello World" en C.....	204
présentation de JNI.....	211
JNI: types, accès aux membres, création d'objets.....	212
références sur des objets JAVA:	215
exceptions.....	216
invocation de JAVA dans du C.....	217
Annexe : collections.....	218
généralités	219
Vector (java 1.1).....	220
Hashtable (java 1.1).....	224
Properties	226
Enumeration (java 1.1)	229
Collections en plateforme Java 2.....	230
Collections : dictionnaires, ensembles	231
Ordre "naturel", comparateurs.....	232
Classes de service : Collections , Arrays.....	233
Itérateurs	234
Annexe : les composants AWT.....	236
List.....	241
TextArea	244
Frame	246
Panel.....	247
Dialog.....	248
FileDialog	250
ScrollPane.....	251
Menus	252
MenuBar	253
Menu	254
MenuItem.....	255
CheckboxMenuItem	256
PopupMenu	257
Contrôle des aspects visuels.....	259
Impression.....	262



Annexe : l'évolution des APIs JAVA	264
Graphique, Multimedia.....	265
Réseau.....	266
Utilitaires, composants.....	267
Utilitaires programmation.....	268
Accès données	269
Echanges sécurisés	270
Embarqué léger	271
Système.....	272
Produits divers	274
Java Enterprise APIs	275

Sun Microsystems France 278



points essentiels :

Rappels :

- Les **Applets** constituent des petites applications java hébergées au sein d'une page html, leur code est téléchargé par le navigateur.
- Une Applet est, à la base, un panneau graphique. Un protocole particulier le lie au navigateur qui le met en oeuvre (cycle de vie de l'Applet).
- Le code de l'Applet fait éventuellement appel à du code ou à des ressources qui sont situées sur le serveur http. Pour minimiser les échanges navigateur/serveur il est intéressant de mettre les codes et les ressources concernées dans des archives **Jar**.
- Les codes de l'Applet sont soumis à des restrictions de sécurité.
- Il est également possible d'utiliser des archives jar et les mécanismes de sécurité pour des applications autonomes.



Applets : lancement

Syntaxe des balises HTML :

```
<APPLET
  [archive=ListeArchives]
  code=package.NomApplet.class
  width=pixels height=pixels
  [codebase=codebaseURL]
  [alt=TexteAlternatif]
  [name=nomInstance]
  [align=alignement]
  [vspace=pixels] [hspace=pixels]
>
  [<PARAM name=Attribut1 value=value>]
  [<PARAM name=Attribut2 value=value>]
  . . . . .
  [alternateHTML]
</APPLET>
```

Exemple :

```
<APPLET
  code=fr.acme.MonApplet.class
  width=300 height=400
>
</APPLET>
```

Evolutions futures (HTML 4) :

```
<OBJECT codetype="application/java"
  classid="fr.acme.MonApplet.class"
  width=300 height=400>
>
</OBJECT>
```

Applets : lancement

Au sein d'un document HTML une Applet est une zone graphique gérée par un programme téléchargé par le navigateur. La balise HTML APPLET comporte les attributs suivant :

- `code=appletFile.class` - Cet attribut *obligatoire* fournit le nom du fichier contenant la classe compilée de l'applet (dérivée de `java.applet.Applet`). Son format pourrait également être `aPackage.appletFile.class`.

Note – La localisation de ce fichier est relative à l'URL de base du fichier HTML de chargement de l'applet.

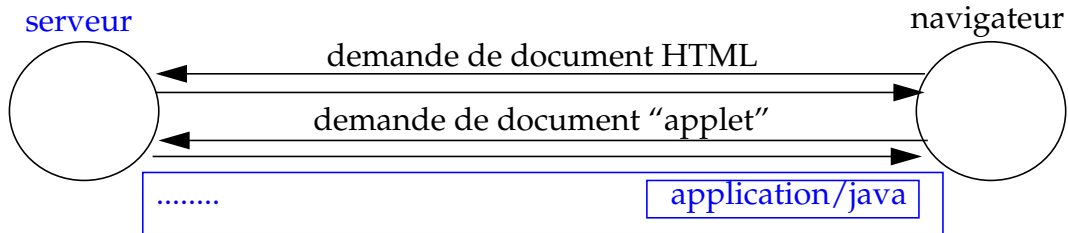
- `width=pixels height=pixels` - Ces attributs *obligatoires* fournissent la largeur et la hauteur initiales (en pixels) de la zone d'affichage de l'applet, sans compter les éventuelles fenêtres ou boîtes de dialogue affichées par l'Applet.
- `codebase=codebaseURL` - Cet attribut facultatif indique l'URL de base de l'applet : le répertoire contenant le code de l'applet. Si cet attribut n'est pas précisé, c'est l'URL du document qui est utilisé.
- `name=appletInstanceName` -- Cet attribut, facultatif, fournit un nom pour l'instance de l'applet et permet de ce fait aux applets situées sur la même page de se rechercher mutuellement (et de communiquer entre-elles).
- `archive=ListeArchives` permet de spécifier une liste de fichiers archive .jar contenant les classes exécutables et, éventuellement des ressources. Les noms des archives sont séparés par des virgules.
- `object=objectFile.ser` permet de spécifier une instance d'objet à charger.

`<param name=appletAttribute1 value=value>` -- Ces éléments permettent de spécifier un paramètre à l'applet. Les applets accèdent à leurs paramètres par la méthode `getParameter()`.

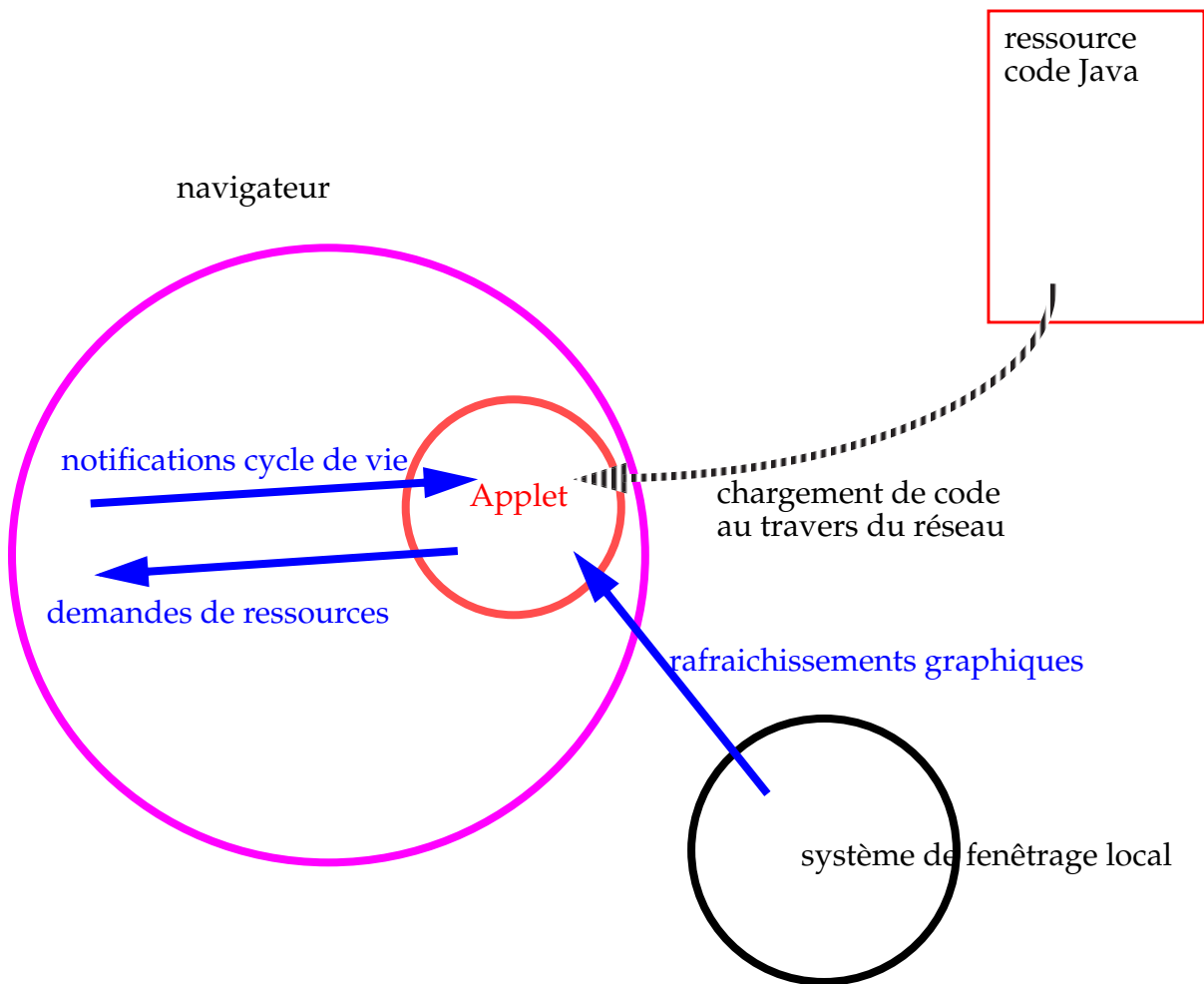


Applets: protocoles

Les échanges HTTP correspondant à une demande d'Applet :



Les APIs entre l'Applet et son environnement :



Applets: protocoles

Une Applet hérite de la classe `java.applet.Applet` qui, elle même, dérive de `java.awt.Panel` qui décrit un panneau graphique (ceci dit une Applet n'est pas nécessairement un objet graphique).

Ce qui caractérise le programme qui s'exécute en tant qu'Applet est qu'il n'a pas de point d'entrée (*main*), qu'il est chargé et lancé par le navigateur et que ses interactions potentielles avec l'environnement se font selon trois catégories d'APIs:

- Les méthodes qui décrivent le cycle de vie de l'Applet.
`init()`, `start()`, `stop()`, `destroy()`
Ces méthodes sont appelées par le navigateur pour notifier à l'Applet certains événements comme l'initialisation de l'Applet, l'icônisation de la page HTML hôte, etc.
Par défaut ces méthodes ne font rien et il faut en redéfinir certaines d'entre elles pour obtenir un comportement de l'Applet.
- Les méthodes qui permettent à l'Applet d'obtenir du navigateur des informations sur la page HTML courante, ou d'obtenir la recherche et le chargement de ressources (distantes).
- Les méthodes qui sont appelées par le système graphique pour la notification de demande de rafraîchissement d'une zone de l'écran:
`repaint()`, `update(Graphics)`, `paint(Graphics)`
La redéfinition éventuelle de certaines de ces méthodes permet à l'Applet d'agir, à un bas niveau, sur son aspect graphique.



Applets : cycle de vie

init () : Cette méthode est appelée au moment où l'applet est créée et chargée pour la première fois dans un navigateur activé par Java (comme AppletViewer). L'applet peut utiliser cette méthode pour initialiser les valeurs des données. Cette méthode n'est pas appelée chaque fois que le navigateur ouvre la page contenant l'applet, mais seulement la première fois juste après le changement de l'applet.

La méthode **start ()** est appelée pour indiquer que l'applet doit être "activée". Cette situation se produit au démarrage de l'applet, une fois la méthode `init()` terminée. Elle se produit également lorsque le navigateur est restauré après avoir été iconisé ou lorsque la page qui l'héberge redevient la page courante du navigateur. Cela signifie que l'applet peut utiliser cette méthode pour effectuer des tâches comme démarrer une animation ou jouer des sons.

```
public void start() {  
    musicClip.play();  
}
```

La méthode **stop ()** est appelée lorsque l'applet cesse de "vivre". Cette situation se produit lorsque le navigateur est icônisé ou lorsque le navigateur présente une autre page que la page courante. L'applet peut utiliser cette méthode pour effectuer des tâches telles que l'arrêt d'une animation.

```
public void stop() {  
    musicClip.stop();  
}
```

Les méthodes `start ()` et `stop ()` forment en fait une paire, de sorte que `start ()` peut servir à déclencher un comportement dans l'applet et `stop ()` à désactiver ce comportement.

destroy () : Cette méthode est appelée avant que l'objet applet ne soit détruit c.a.d enlevé du cache du navigateur.

Applets : demandes au navigateur

L'applet peut demander des ressources (généralement situées sur le réseau). Ces ressources sont désignées par des URLs (classe `java.net.URL`). Deux URL de référence sont importantes :

- l'URL du document HTML qui contient la description de la page courante. Cette URL est obtenue par `getDocumentBase()` .
- l'URL du code "racine" de l'Applet (celui qui est décrit par l'attribut "code"). Cette URL est obtenue par `getCodeBase()` .

En utilisant une de ces URLs comme point de base on peut demander des ressources comme des images ou des sons :

- `getImage(URL base, String désignation)` : permet d'aller rechercher une image, rend une instance de la classe `Image`.
- `getAudioClip(URL base, String désignation)` : permet d'aller rechercher un son, rend une instance de la classe `AudioClip`.



Les désignations de ressource par rapport à une URL de base peuvent comprendre des chemins relatifs (par ex: `"../../images/truc.gif"`). Attention toutefois : certains systèmes n'autorisent pas des remontées dans la hiérarchie des répertoires.

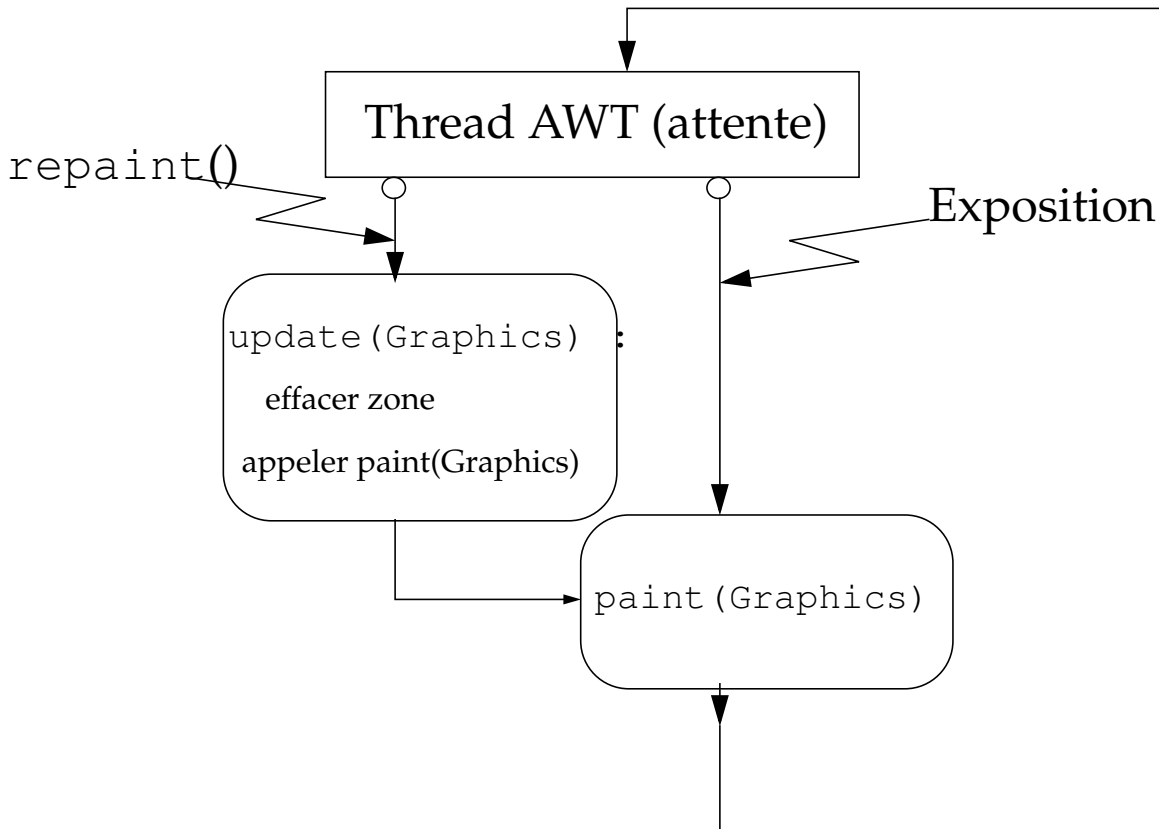
Le moteur son de la plateforme Java2 sait traiter des fichiers `.wav`, `.aiff` et `.au` ainsi que des ressource MIDI. Une nouvelle méthode `newAudioClip(URL)` permet de charger un `AudioClip`.

La méthode `getParameter(String nom)` permet de récupérer, dans le fichier source HTML de la page courante, la valeur d'un des éléments de l'applet courante, décrit par une balise `<PARAM>` et ayant l'attribut `name=nom` . Cette valeur est une chaîne `String`.



Applets : demandes du système de fenêtrage

Les mises à jour du système graphique de bas niveau :



- **repaint ()** : demande de rafraîchissement de la zone graphique. Cette demande est asynchrone et appelle `update(Graphics)`. L'instance de `java.awt.Graphics` donne un contexte graphique qui permet aux méthodes de dessin d'opérer.
- **update(Graphics)** : par défaut efface la zone et appelle `paint(Graphics)`. Cette méthode peut-être redéfinie pour éviter des papillotements de l'affichage (séries d'effacement/dessin).
- **paint(Graphics)** : la redéfinition de cette méthode permet de décrire des procédures logiques de (re)construction des dessins élémentaires qui constituent la présentation graphique. Lorsque l'on utilise des composants graphiques de haut niveau il n'est souvent pas nécessaire d'intervenir et le système AWT gère automatiquement ces (re)affichages.

Applets : exemple

```
// Suppose l'existence de "sounds/cuckoo.au"
// à partir du répertoire du fichier HTML d'origine

import java.awt.Graphics;
import java.applet.*;

public class HwLoop extends Applet {
    AudioClip sound;

    public void init() {
        sound = getAudioClip(getDocumentBase(),
                             "sounds/cuckoo.au");
    }

    public void paint(Graphics gr) {
        // méthode de dessin de java.awt.Graphics
        gr.drawString("Audio Test", 25, 25);
    }

    public void start () {
        sound.loop();
    }

    public void stop() {
        sound.stop();
    }
}
```



Applets : utilisation d'archives Jar

Chaque fois qu'une Applet a besoin du code d'une autre classe située sur le site serveur ou chaque fois qu'elle a besoin de charger une ressource il y a une transaction HTTP entre le navigateur et le serveur.

Il peut s'avérer nécessaire de diminuer ce nombre de transactions en regroupant l'ensemble de ces ressources dans une (ou plusieurs) **archives jar**.

Les archives Jar permettent de regrouper dans un fichier compressé (format ZIP) un ensemble de répertoires. Des meta-informations sont présentes dans un fichier `Manifest`.

Création d'une archive à partir du répertoire "repClasses" et contenant tout le package "fr.acme.applets"

```
jar cvf MesApplets.jar -C repClasses fr/acme/applets
```

Mise à jour du contenu avec les ressources du répertoire "images"

```
jar uvf MesApplets.jar images
```

L'archive Jar contiendra alors sous sa "racine" :

- le répertoire `fr` (sous lequel se trouvent les codes des classes)
- le répertoire `images` (qui contient les images)
- le répertoire `META-INF` (qui contient les meta-informations sur les ressources contenues dans le jar: fichier `MANIFEST.MF` .

Pour connaître le contenu d'un jar :

```
jar tvf MesApplets.jar
```

Applets : Ressources dans une archive Jar

Pour exploiter l'archive Jar générée dans l'exemple précédent :

Extrait du HTML :

```
<APPLET archive=MesApplets.jar code=fr.acme.applets.MonApplet.class
width=200 height=300>
  <PARAM name=image1 value=/images/duke.gif>
</APPLET>
```

Code java de MonApplet:

```
Image im ;
public void init() {
    Class maClasse = this.getClass() ;
    // ou fr.acme.applets.MonApplet.class
    im = getImage(maClasse.getResource(
        getParameter("image1")));
}

public void paint(Graphics gr) {
    gr.drawImage(im, 0, 0, this) ;
}
```

La méthode `getResource` s'adresse au `ClassLoader` qui a chargé la classe courante pour obtenir l'URL d'une ressource.



Le système de sécurité de Java

Sauf indication contraire de l'utilisateur du navigateur une Applet est considérée comme non fiable et ne peut :

- obtenir des informations sur l'environnement local (nom de l'utilisateur, répertoire d'accueil, etc.). Seules des informations sur le type de système et de JVM sont accessibles.
- connaître ou ouvrir des fichiers dans le système local, lancer des tâches locales.
- obtenir une connexion sur le réseau avec une adresse autre que celle du site d'origine de la page HTML (une Applet peut donc ouvrir une connexion avec son hôte d'origine).

La classe **AccessControler** est chargée des contrôles de sécurité. Il est possible de modifier les droits d'accès en définissant une politique de sécurité. Les droits sont accordés à des codes (en les désignant par leur URL) et sont décrits dans des ressources locales.

Exemple de politique de site dans fichier "\$JAVA_HOME/lib/security/java.policy" :

```
grant codeBase "file:${java.home}/lib/ext/-" {  
    permission java.security.AllPermission;  
};  
//illustre les droits des "extensions installées"
```

Politique propre à un utilisateur dans "\$HOME/.java.policy":

```
grant codeBase "http://www.acme.fr" {  
    permission java.io.FilePermission "/tmp/*", "read" ;  
}
```



Attention : la notation de codeBase est celle d'une URL (avec séparateur "/"), la notation du nom de fichier dans FilePermission est locale au système (utiliser la variable "\${/}" pour avoir un séparateur portable)

Applications autonomes: sécurité

Il est possible de déclencher les contrôles de sécurité sur une application autonome en la lançant avec un SecurityManager:

```
java -Djava.security.manager MonApplication
```

Dans le CLASSPATH la sécurité distingue les classes "système" fiables (propriété `java.sys.class.path`) des autres classes soumises aux contrôles de sécurité (propriété `java.class.path`).

Dès qu'une application autonome met en oeuvre un SecurityManager il est vivement conseillé de personnaliser la politique de sécurité qui s'applique spécifiquement à cette application.

On peut passer un fichier de description de politique de sécurité au lancement d'une application :

```
java -Djava.security.policy=monfichier.policy MonApplication
```

Les descriptions contenues dans le fichier "monfichier.policy" complètent alors la politique de sécurité courante.

Il est également possible de redéfinir (et remplacer) complètement la politique courante en utilisant une autre syntaxe :

```
java -Djava.security.policy==total.policy MonApplication
```



Applications autonomes : archives Jar

Par ailleurs il est également possible de mettre une application autonome dans un fichier jar. Il faut créer le fichier jar en initialisant l'entrée "Main-Class" du fichier **Manifest**. Exemple : soit le fichier *monManifeste* :

```
Main-Class: fr.acme.MonApplication
```

Création du fichier jar :

```
jar cvfm appli.jar monManifeste -C classes fr/acme
```

Lancement de l'application (ici sans gestionnaire de sécurité):

```
java -jar appli.jar
```

Approfondissements

* Pour les opérations graphiques de “bas niveau” voir :

- L'utilisation de la Classe `java.awt.MediaTracker`. Le chargement des images (par exemple) se fait de manière asynchrone. Si, pour manipuler des images, on a besoin d'attendre leur chargement complet, utiliser un `MediaTracker`.
- Les classes `java.awt.Graphics` et `java.awt.Graphics2D`
- Notre séminaire java media (SL-053)

* Pour plus d'informations sur les archives JAR voir la documentation dans “[docs/guide/jar/index.html](#)”. Pour des opérations sur ces fichiers voir le package “`java.util.jar`” (à approfondir après maîtrise des mécanismes d'entrée/sortie)

* Pour des approfondissements sur la sécurité et en particulier sur la “signature” des archives Jar voir “[docs/guide/security/index.html](#)”, voir également nos cours SL-051 et SL-303.





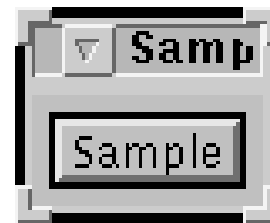
points essentiels

Le package AWT concerne les services du terminal virtuel portable:

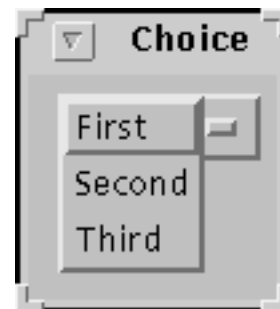
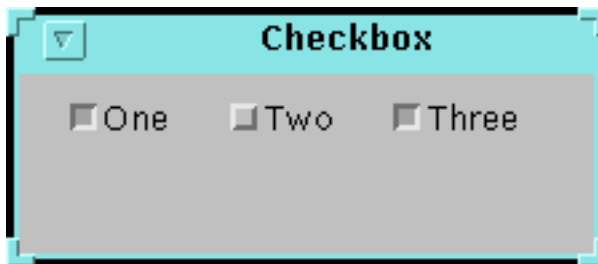
- Les composants AWT s'appuient sur des services "natifs" du système de fenêtrage local. Ainsi par exemple à un "Button" (bouton) correspondra un composant natif bouton-Motif, bouton-Windows, etc.
Un autre package (Swing) permet de disposer de composants graphiques 100% pur java.
- Les composants principaux sont des `Components`; certains d'entre eux sont aussi des `Containers`, c'est à dire des composants qui peuvent en contenir d'autres.
- La disposition des `Components` à l'intérieur des `Containers` est sous le contrôle de gestionnaires de disposition (`LayoutManager`).



Accès aux manipulations graphiques



Button



Accès aux manipulations graphiques

Le package AWT

Le package AWT fournit les objets pour accéder aux services du terminal virtuel portable.

Une notion fondamentale dans ce package est la hiérarchie `Component` (un objet graphique) et `Container` (dérivé du précédent: on peut disposer plusieurs `Components` DANS un `Container`)

Exemples de `Container` : `Frame` (une fenêtre), `Panel` (un "panneau") et son dérivé particulier qu'est l'`Applet`.

Exemples de `Component`: `Button`, `Label` (un étiquette), `TextField` (une zone de saisie Texte), `Canvas` (zone de dessin graphique).



Les gestionnaires de Disposition (LayoutManager)

Un des points forts de JAVA est de permettre de faire exécuter le même programme sur des plateformes différentes sans en modifier le code. Pour les interactions graphiques une des conséquences de cette situation est qu'un même programme va devoir s'afficher sur des écrans ayant des caractéristiques très différentes. On ne peut donc raisonnablement s'appuyer sur un positionnement des composants en absolu (avec des coordonnées X et Y fixes).

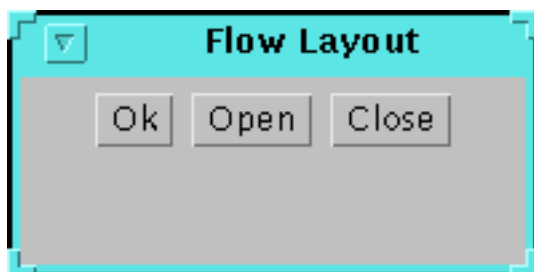
La disposition relative des différents composants à l'intérieur d'un `Container` sera prise en charge par un "gestionnaire de disposition" attaché à ce container. Ce `LayoutManager` va savoir gérer les positions des composants en fonctions des déformations subies par le `Container` correspondant.

A chaque `Container` est associé une liste des composants contenus. Attention une instance de composant ne peut être disposée qu'à UN SEUL endroit (il ne sert à rien de faire plusieurs opérations `add()` avec le même composant -sauf si on veut explicitement le faire changer de zone d'affichage-)

FlowLayout :

FlowLayout : dispose les composants "en ligne". C'est le gestionnaire par défaut des Panels.

exemple de FlowLayout



La disposition se fait par une série d'appels à `add(Component)` (l'ordre des appels est important puisqu'il détermine les positions relatives des composants)

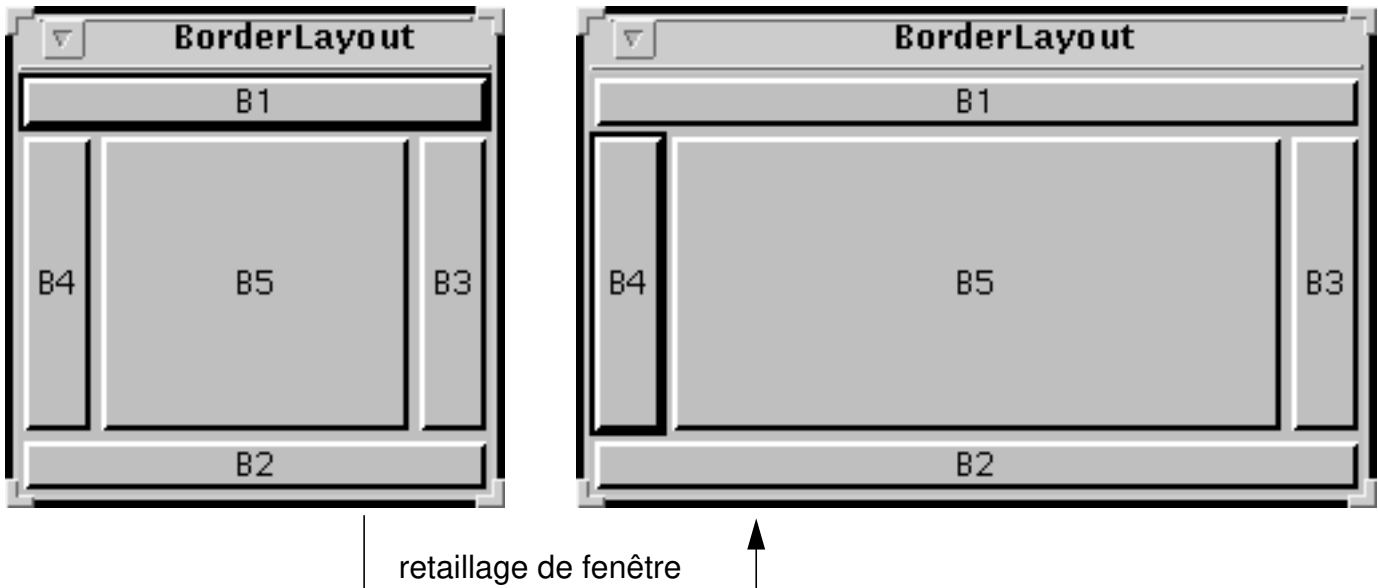
```
panneau.add(new Button("bouton 1")) ;  
panneau.add(new Button("bouton 2")) ;
```



BorderLayout :

BorderLayout : dispose des zones dans des points cardinaux autour d'une zone centrale qui tend à occuper la plus large place possible. C'est le gestionnaire par défaut des Frames.

Exemple de BorderLayout :



Les appels à `add()` sont "qualifiés" (on désigne l'emplacement dans le zonage particulier au gestionnaire). Les composants ajoutés sont souvent des Containers ayant eux-même leur propre gestionnaire de disposition.

Exemple:

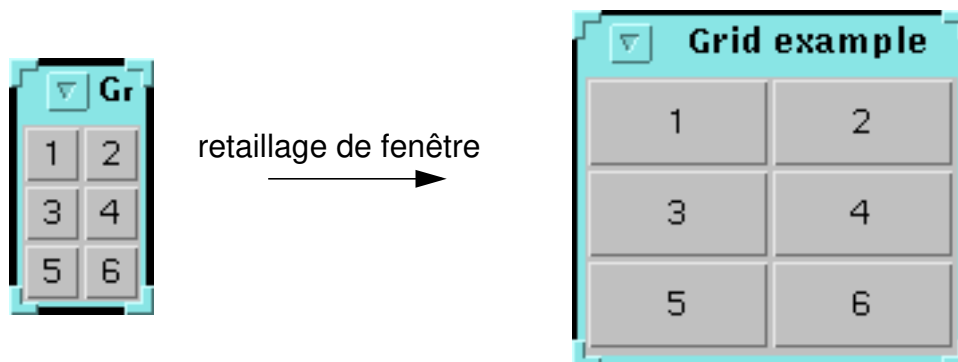
```
Panel panCentral = new Panel() ;
Panel panBas = new Panel() ;

fenêtre.add(panCentral, BorderLayout.CENTER) ;
fenêtre.add(panBas, BorderLayout.SOUTH) ;

// le panneau bas a son propre gestionnaire
panbas.add(new Button ("OK")) ;
```

GridLayout :

GridLayout : permet de disposer des composants dans une grille. Toutes les cellules de la grille prennent la même taille et contraignent les composants qu'elles contiennent à se déformer.



On peut utiliser deux constructeurs différents par ex :

```
setLayout(new GridLayout(int rows, int cols);
```

ou

```
setLayout(new GridLayout(
    int rows, int cols, int hgap, int vgap);
```

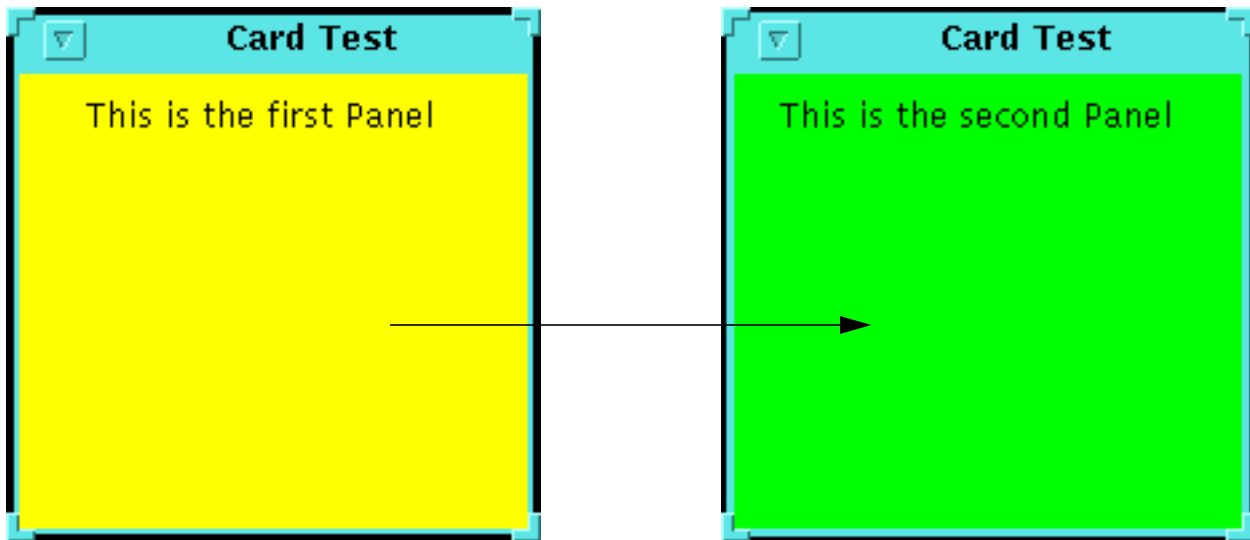
L'ordre des appels à `add()` est important puisqu'il détermine les positions relatives des composants :

```
fenêtre.setLayout(new GridLayout(0,2)) ;
fenêtre.add(new Button("1")) ;// position 0 0
fenêtre.add(new Button("2")) ; // position 0 1
fenêtre.add(new Button("3")) ;// position 1 0
...
```



CardLayout :

CardLayout : permet de disposer des composants dans une "pile" seul le composant du dessus est visible et on dispose de méthodes spéciales pour faire passer un composant particulier sur le "dessus" de la pile.



Il faut écrire un programme permettant de passer d'un panneau à l'autre...

Les appels à `add()` sont "qualifiés" (on donne dynamiquement un nom au niveau occupé par le composant, ce nom sera utilisé pour rechercher ultérieurement ce niveau). Les composants ajoutés sont souvent des Containers ayant eux-même leur propre gestionnaire de disposition.

```
CardLayout cartes = new CardLayout() ;
fen.setLayout(cartes) ;
fen.add(panel1, "Un") ;
fen.add(panel2, "Deux") ;
....
cartes.show(fen, "Un") ;

;
```


GridBagLayout :

GridBagLayout : dispose les composants à l'intérieur des "cellules" d'une table. Chaque ligne ou colonne de la table peut avoir des dimensions différentes de celles des autres lignes ou colonnes (quadrillage irrégulier).



Les paramètres contrôlant la mise en place d'un composant particulier sont décrits par une instance de la classe `GridBagConstraints` (on peut utiliser sans risque la même instance pour plusieurs composants)

- a. *gridx, gridy* : donne les coordonnées x, y de l'objet dans la grille (celle-ci déduit automatiquement son propre nombre de lignes et de colonnes)
- b. *gridwidth, gridheight* : nombre de cellules occupées par le composant
- c. *fill* : direction du remplissage (le composant tend alors à occuper toute sa cellule dans la direction donnée). Valeurs: NONE, BOTH, VERTICAL, HORIZONTAL
- d. *anchor*: lorsqu'un composant est plus petit que sa cellule, bord d'ancrage du composant (un point cardinal: EAST, NORTHEAST, etc..)
- e. *insets*: détermination des "gouttières" (distance minimum entre le composant et les frontières de sa cellule)
- f. *weightx, weighty* : "poids" relatif de la cellule (valeur de type `double` comprise entre 0 et 1)



- g. *ipadx, ipady* : Signale le remplissage interne, à savoir la quantité à ajouter à la taille minimale du composant. La largeur minimale du composant est la somme de la largeur minimale et de $ipadx*2$ pixels (puisque le remplissage s'applique dans les deux sens). Dans la même logique, la hauteur minimale du composant est la somme de la hauteur minimale et de $ipady*2$ pixel

```
// on a un tableau a deux dimensions comprenant des composants
Component[][] tb = {
    {...},
    {...},
    ...
};

this.setLayout(new GridBagLayout() );
GridBagConstraints gbc = new
    GridBagConstraints (0,0, 1,1, // x,y, w, h
        1.0, 1.0, // weightx,y
        GridBagConstraints.WEST , //anchor
        GridBagConstraints.NONE , //fill
        new Insets(3,5,3,5) ,
        0,0 ); //ipadx, ipady

for (int iy = 0 ; iy <tb.length; iy ++ ) {
    gbc.gridy= iy ;
    for (int ix = 0 ; ix< tb[iy].length ; ix++) {
        gbc.gridx = ix ;
        if( null != tb[iy][ix] ) {
            add(tb[iy][ix], gbc) ;
        }
    }
}
}
```

Approfondissements

* Voir en annexe quelques composants AWT courants.

* Les primitives graphiques de dessin sont liées aux classes Graphics et Graphics2D

* Les primitives graphiques peuvent être utilisées pour "décorer" des composants d'un type prédéfini ou même pour créer des "Composants poids-plume" en agissant sur des objets créés en sous-classant directement Component ou Container. De tels objets graphiques sont initialement transparents et ne sont pas associés à des objets natifs du système de fenêtrage local, il faut gérer par programme l'ensemble de leur comportement (aspects, événements).

De tels objets constituent l'essentiel de bibliothèques d'objets d'interaction comme les composants SWING





Points essentiels :

Le traitement des événements permet d'associer des comportements à des présentations AWT :

- Le modèle de gestion des événements a changé entre les versions 1.0 et 1.1 de Java.
- A partir de la version 1.1 il faut associer un gestionnaire d'événement à un composant sur lequel on veut surveiller un type donné d'événement.
- A chaque type d'événement correspond un contrat d'interface.
- Lorsque la réalisation de ce contrat d'interface conduit à un code trop bavard on peut faire dériver le gestionnaire d'événement d'une classe "Adapter".



Les événements

Lorsque l'utilisateur effectue une action au niveau de l'interface utilisateur, un *événement* est émis. Les événements sont des objets qui décrivent ce qui s'est produit. Il existe différents types de classes d'événements pour décrire des catégories différentes d'actions utilisateur.

Evénements sources

Un événement source (au niveau de l'interface utilisateur) est le résultat d'une action utilisateur sur un composant AWT. A titre d'exemple, un clic de la souris sur un composant bouton génère (source) un `ActionEvent`. L'`ActionEvent` est un objet (une instance de la classe) contenant des informations sur le statut de l'événement :

- `ActionCommand` : nom de commande associé à l'action.
- `modifiers` : tous modificateurs mobilisés au cours de l'action.

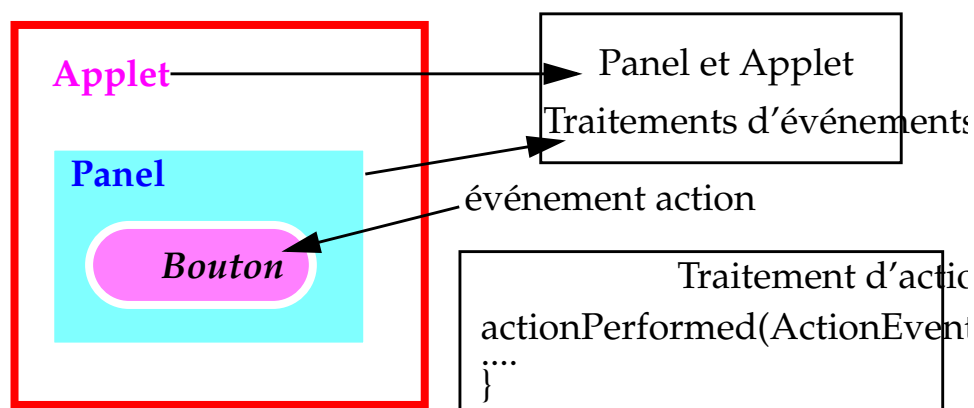
Traitements d'événements

Lorsqu'un événement se produit, ce dernier est reçu par le composant avec lequel l'utilisateur interagit (par exemple un bouton, un curseur, un `textField`, etc.). Un traitement d'événement est une méthode qui reçoit un objet `Event` de façon à ce que le programme puisse traiter l'interaction de l'utilisateur.

Modèle d'événements JDK 1.1

Modèle de délégation (JDK 1.1)

JDK 1.1 a introduit un nouveau modèle d'événement appelé modèle d'événement par délégation. Dans un modèle d'événement par délégation, les événements sont envoyés au composant, mais c'est à chaque composant d'enregistrer une routine de traitement d'événement (appelé *veilleur*: `Listener`) pour recevoir l'événement. De cette façon, le traitement d'événement peut figurer dans une classe distincte du composant. Le traitement de l'événement est ensuite délégué à une classe séparée.





modèle d'événements JDK 1.1

Les événements sont des objets qui ne sont renvoyés qu'aux veilleurs enregistrés. A chaque type d'événement est associé une interface d'écoute correspondante.

A titre d'exemple, voici un cadre simple comportant un seul bouton :

```
import java.awt.*;
public class TestButton {
    public static void main (String args[]){
        Frame f = new Frame ("Test");
        Button b = new Button("Press Me!");
        b.addActionListener(new ButtonHandler());
        f.add(b, BorderLayout.CENTER);
        f.pack();
        f.setVisible(true);
    }
}
```

La classe ButtonHandler définit une instance de traitement de l'événement .

```
import java.awt.event.*;
public class ButtonHandler implements
    ActionListener{
    public void actionPerformed(ActionEvent e) {
        System.out.println("Action occured");
    }
}
```


Modèle d'événements JDK 1.1

Modèle de délégation (JDK 1.1) (suite)

- La classe `Button` comporte une méthode `addActionListener(ActionListener)`.
- L'interface `ActionListener` définit une méthode simple, `actionPerformed` qui recevra un `ActionEvent`.
- Lorsqu'un objet de la classe `Button` est créé, l'objet peut enregistrer un veilleur pour les `ActionEvent` par l'intermédiaire de la méthode `addActionListener`, en précisant la classe d'objets qui implémente l'interface `ActionListener`.
- Lorsque l'on clique sur l'objet Bouton avec la souris, un `ActionEvent` est envoyé à chaque `ActionListener` enregistré par l'intermédiaire de la méthode `actionPerformed (ActionEvent)`.



modèle d'événements JDK 1.1

Modèle de délégation (JDK 1.1) (suite)

Cette approche présente plusieurs avantages :

- Il est possible de créer des classes de filtres pour classer les événements .
- Le modèle de délégation est plus adapté à la répartition du travail entre les classes.
- Le nouveau modèle d'événement supporte Java Beans™.

Certains problèmes/inconvénients du modèle méritent également d'être considérés :

- Il est plus difficile à comprendre, au moins au départ.
- Le passage du code JDK 1.0 au code JDK 1.1 est compliqué.
- Bien que la version actuelle de JDK gère le modèle d'événement JDK 1.0 en plus du modèle de délégation, les modèles d'événements JDK 1.0 et JDK 1.1 ne peuvent pas être mélangés.

Comportement de l'interface graphique utilisateur Java

Catégories d'événements

Le mécanisme général de réception des événements à partir de composants a été décrit dans le contexte d'un seul type d'événement. Plusieurs événements sont définis dans le package `java.awt.event`, et des composants tiers peuvent s'ajouter à cette liste.

Pour chaque catégorie d'événements, il existe une interface qui doit être implémentée par toute classe souhaitant recevoir ces événements. Cette interface exige aussi qu'une ou plusieurs méthodes soient définies. Ces méthodes sont appelées lorsque des événements particuliers surviennent. Le tableau de la page suivante liste les catégories et indique le nom de l'interface correspondante ainsi que les méthodes associées. Les noms de méthodes sont des mnémoniques indiquant les conditions générant l'appel de la méthode.

On remarquera qu'il existe des événements de bas niveau (une touche est pressée, on clique la souris) et des événements abstraits de haut niveau (Action = sur un bouton on a cliqué, sur un TextField on a fait un <retour chariot>, ...)



Tableau des interfaces de veille

Catégorie	Interface	Methodes
Action	ActionListener	<code>actionPerformed (ActionEvent)</code>
Item	ItemListener	<code>itemStateChanged (ItemEvent)</code>
Mouse Motion	MouseMotionListener	<code>mouseDragged (MouseEvent)</code> <code>mouseMoved (MouseEvent)</code>
Mouse	MouseListener	<code>mousePressed (MouseEvent)</code> <code>mouseReleased (MouseEvent)</code> <code>mouseEntered (MouseEvent)</code> <code>mouseExited (MouseEvent)</code> <code>mouseClicked (MouseEvent)</code>
Key	KeyListener	<code>keyPressed (KeyEvent)</code> <code>keyReleased (KeyEvent)</code> <code>keyTyped (KeyEvent)</code>
Focus	FocusListener	<code>focusGained (FocusEvent)</code> <code>focusLost (FocusEvent)</code>
Adjustement	AdjustmentListener	<code>adjustmentValueChanged (AdjustmentEvt)</code>
Component	ComponentListener	<code>componentMoved (ComponentEvent)</code> <code>componentHidden (ComponentEvent)</code> <code>componentResize (ComponentEvent)</code> <code>componentShown (ComponentEvent)</code>
Window	WindowListener	<code>windowClosing (WindowEvent)</code> <code>windowOpened (WindowEvent)</code> <code>windowIconified (WindowEvent)</code> <code>windowDeiconified (WindowEvent)</code> <code>windowClosed (WindowEvent)</code> <code>windowActivated (WindowEvent)</code> <code>windowDeactivated (WindowEvent)</code>
Container	ContainerListener	<code>componentAdded (ContainerEvent)</code> <code>componentremoved (ContainerEvent)</code>
Text	TextListener	<code>textValueChanged (TextEvent)</code>

Événements générés par les composants AWT

Table 1:

Composant AWT	Acti on	adju st	com pon ent	cont aine r	focu s	item	key	mou se	mou se moti on	text	win dow
Button	●		●		●		●	●	●		
Canvas			●		●		●	●	●		
Checkbox			●		●	●	●	●	●		
CheckboxMenuItem						●					
Choice			●		●	●	●	●	●		
Component			●		●		●	●	●		
Container			●	●	●		●	●	●		
Dialog			●	●	●		●	●	●		●
Frame			●	●	●		●	●	●		●
Label			●		●		●	●	●		
List	●		●		●	●	●	●	●		
MenuItem	●										
Panel			●	●	●		●	●	●		
Scrollbar		●	●		●		●	●	●		
ScrollPane			●	●	●		●	●	●		
TextArea			●		●		●	●	●	●	
TextComponent			●		●		●	●	●	●	
TextField	●		●		●		●	●	●	●	
Window			●	●	●		●	●	●		●



Détails sur les mécanismes

Obtention d'informations sur un événement

Lorsque les méthodes de traitement, telles que `mouseDragged()` sont appelées, elles reçoivent un argument qui peut contenir des informations importantes sur l'événement initial. Pour savoir en détail quelles informations sont disponibles pour chaque catégorie d'événement, reportez-vous à la documentation relative à la classe considérée dans le package `java.awt.event`.

Récepteurs multiples

La structure d'écoute des événements AWT permet actuellement d'associer plusieurs veilleurs au même composant. En général, si on veut écrire un programme qui effectue plusieurs actions basées sur un même événement, il est préférable de coder ce comportement dans la méthode de traitement.

Cependant, la conception d'un programme exige parfois que plusieurs parties non liées du même programme réagissent au même événement. Cette situation peut se produire si, par exemple, un système d'aide contextuel est ajouté à un programme existant.

Le mécanisme d'écoute permet d'appeler une méthode `add*Listener` aussi souvent que nécessaire en spécifiant autant d'écouteurs différents que la conception l'exige. Les méthodes de traitement de tous les écouteurs enregistrés sont appelées lorsque l'événement survient.



L'ordre d'appel des méthodes de traitement n'est pas défini. En général, si cet ordre a une importance, les méthodes de traitement ne sont pas liées et on ne doit pas utiliser cette fonction pour les appeler. Au lieu de cela, il faut enregistrer simplement le premier écouteur et faire en sorte qu'il appelle directement les autres. C'est ce qu'on appelle un multiplexeur d'événements

Adaptateurs d'événements

Il est évident que la nécessité d'implanter toutes les méthodes de chaque interface d'écouteur représente beaucoup de travail, en particulier pour les interfaces `MouseListener` et `ComponentListener`.

A titre d'exemple, l'interface `MouseListener` définit les méthodes suivantes :

- `mouseClicked (MouseEvent)`
- `mouseEntered (MouseEvent)`
- `mouseExited (MouseEvent)`
- `mousePressed (MouseEvent)`
- `mouseReleased(MouseEvent)`

Pour des questions pratiques, Java fournit une classe d'adaptateurs pour pratiquement chaque interface de veiller, cette classe implante l'interface appropriée, mais ne définit pas les actions associées à chaque méthode.

De cette façon, la routine d'écoute que l'on définit peut hériter de la classe d'adaptateurs et ne surcharger que des méthodes choisies.



Adaptateurs d'événements

Par exemple :

```
import java.awt.*;
import java.awt.event.*;

public class MouseClickHandler extends MouseAdapter {

    //Nous avons seulement besoin du traitement mouseClicked,
    //nous utilisons donc l'adaptateur pour ne pas avoir à
    //écrire toutes les méthodes de traitement d'événement

    public void mouseClicked (MouseEvent e) {
        //Faire quelque chose avec le clic de la souris . . .
    }
}
```


Approfondissements

* Il y a deux manières principales d'assurer les services de surveillance d'événements :

- soit un conteneur englobant assure le service de Listener et c'est sa référence qui est passée au `addXXXListener` correspondant.
- soit on crée une classe spécifique qui peut être une classe interne et, éventuellement, une classe anonyme.

Les deux techniques ont des avantages et des inconvénients : la seconde permet une meilleure indépendance des composants les uns par rapport aux autres, la première est parfois plus facile à mettre en oeuvre.

* La création de composants poids-plume s'accompagne d'une gestion des événements (voir `enableEvents()` et `processEvent()` de `Component`, etc.).





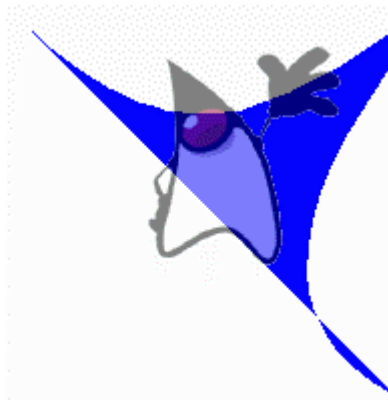
Points Essentiels :

La plateforme Java 2 intègre de nouveaux services pour réaliser des présentations graphiques :

- Java2D permet de réaliser des opérations graphiques sophistiquées et Swing constitue un ensemble cohérent de services et de composants d'interactions graphiques.
- La hiérarchie des composants comprend d'un coté des composants "racine" comme JFrame qui dérivent de composants "lourds" comme Frame, et d'autre part des composants "légers" écrits en Java qui dérivent de la classe JComponent.
- Les deux types de classes offrent de nombreux services spécifiques qui les distinguent des composants AWT classiques.



Java Foundation Classes



Java Foundation Classes

Outre AWT et ses sous-packages les JFC (Java Foundation Classes) comprennent les API suivantes :

- **Java2D** ; complète les primitive graphiques avec des mécanismes de dessin sophistiqués (formes complexes, textures, superpositions de dessins avec transparence, etc.)
Voir `java.awt.Graphics2D`, et les sous-packages de `awt` : `geom`, `image`, `color`, `font`, `print`.
- **l'API d'accessibilité** (`javax.accessibility`) : interfaces pour des technologies d'aide aux interactions pour des personnes handicapées (lecteurs d'écran, magnifieurs, etc.)
- **Drag & Drop** (`java.awt.dnd`, `java.awt.datatransfer`) : permet le coupé/collé et de l'échange de données entre applications.
- **Swing** : offre une bibliothèque de composants et des mécanismes plus riches que ceux de l'AWT standard. La portabilité de ces composants est assurée par le fait que ce sont essentiellement des composants écrits en Java : on n'a donc pas l'obligation d'avoir un composant "natif" correspondant dans le système de fenêtrage local. D'un point de vue organisationnel la mise en place de ces composant permet l'utilisation de paradigmes plus riches (modèle vue-contrôleur) et permet de décharger le programmeur de certaines tâches d'entretien (double-buffering, etc).

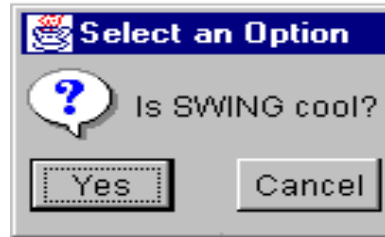


Les Composants SWING

Quelques composants SWING



JButton



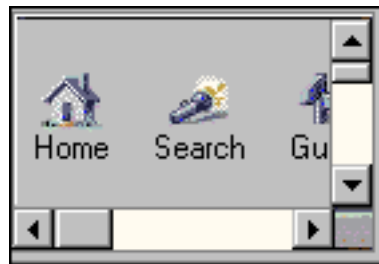
JOptionPane



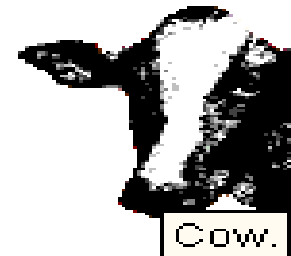
JLabel



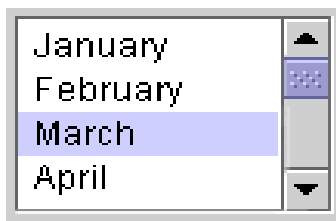
JSlider



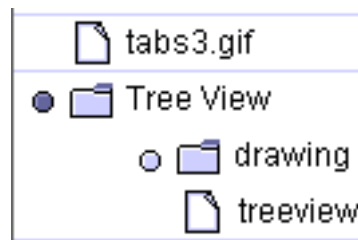
JScrollPane



JToolTip



JList



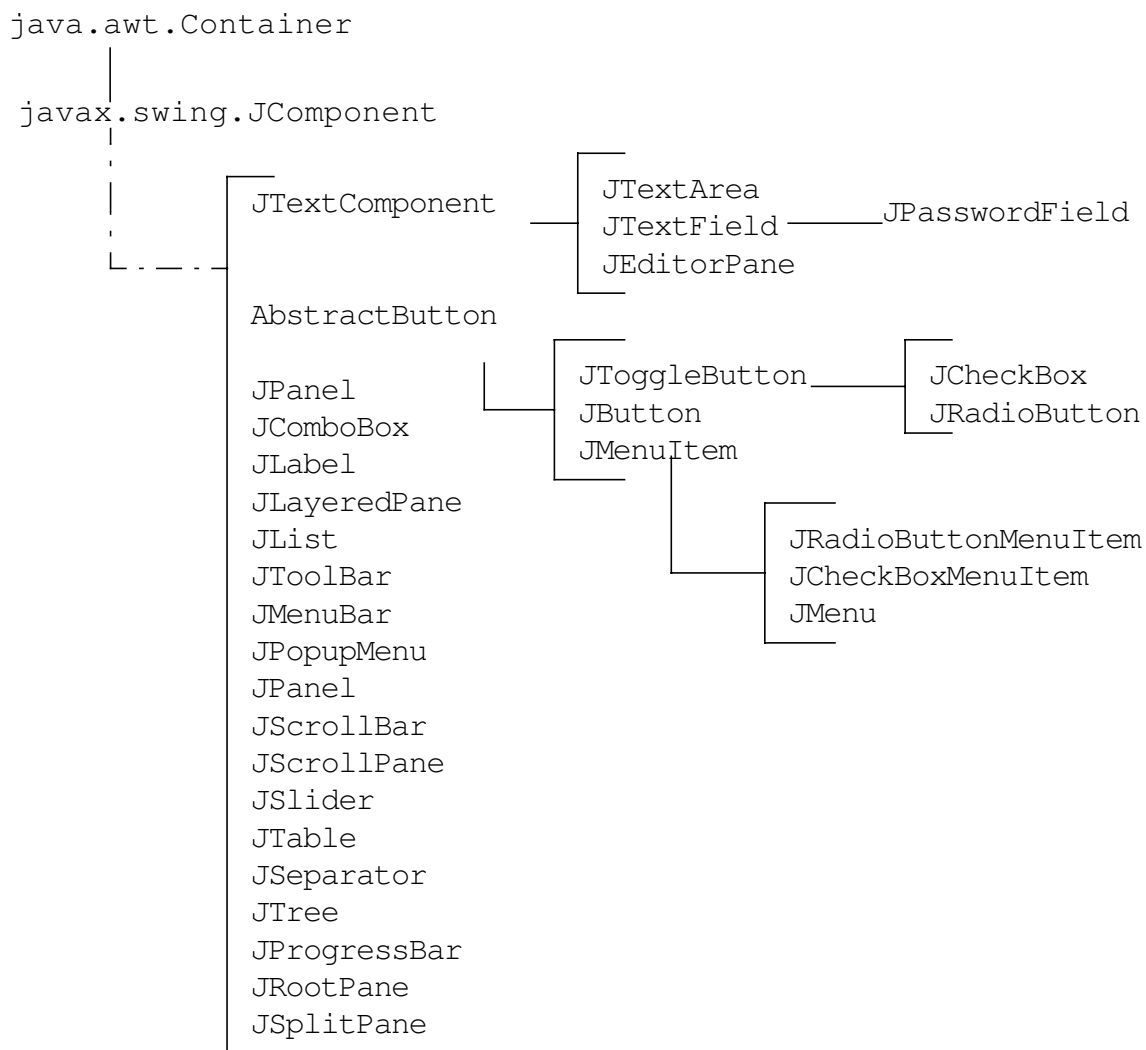
JTree

First Na...	Last Name
Mark	Andrews
Tom	Ball
Alan	Chung
Jeff	Dinkins

JTable

Ces composants sont bâtis à partir de `JComponent` qui dérive de la classe `AWT Container`. Ceci permet des combinaisons complexes et permet d'enrichir ces composants de nombreuses décorations.

SWING : hiérarchie des composants



Les composants SWING et les classes qui les accompagnent sont situés dans le package `javax.swing` et ses sous-packages.



Une application Swing de base :

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class HelloSwing extends JFrame implements ActionListener {
    // en fait un JPanel
    private JComponent contentPane = (JComponent) getContentPane();
    private JLabel jLabel;
    private JButton jButton;

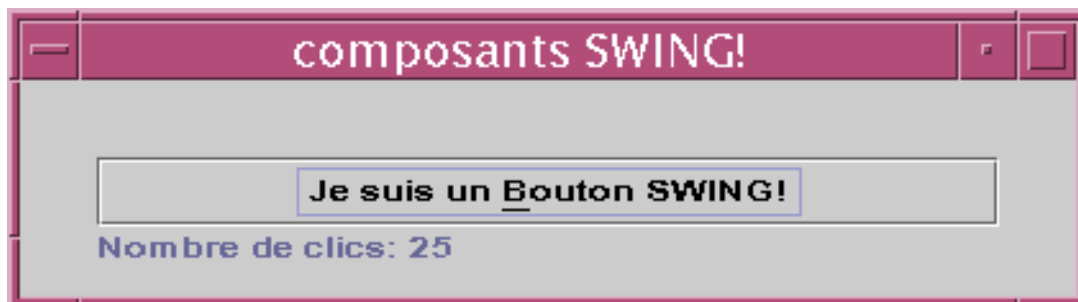
    private String labelPrefix = "Nombre de clics: ";
    private int numClicks = 0;

    public static void main(String[] args) {
        HelloSwing helloSwing = new HelloSwing("composants SWING!");
        helloSwing.init();
        helloSwing.start();
    } // main

    public HelloSwing(String message) {
        super(message) ;
        // que faire si on "ferme" la fenetre (WindowConstants)
        this.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
        // et pour faire bonne mesure on arrête le programme
        // avec une petite classe anonyme
        this.addWindowListener( new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    }

    public void start() {
        this.pack();
        this.setVisible(true);
    } // start()

    // traitement événements bouton
    public void actionPerformed(ActionEvent e) {
        numClicks++;
        jLabel.setText(labelPrefix + numClicks);
    }
}
```

```

public void init() {

    // on peut ici fixer le lookAndFeel
    //
    // try {
    //     UIManager.setLookAndFeel(
    //         UIManager.getLookAndFeel());
    // } catch (UnsupportedLookAndFeelException e) {
    //     System.err.println("Couldn't use the " +
    //         "default look and feel " + e);
    // }

    // le LABEL
    jLabel = new JLabel(
        "cliquez sur le bouton pour déclencher le compteur");

    // le BOUTON
    jButton = new JButton(" Je suis un Bouton SWING! ");
    // un accélérateur ALT-B (même effet qu'un clic souris)
    jButton.setMnemonic('b');
    jButton.addActionListener(this);

    // le CONTAINER
    // on met une bordure autour du "contentPane"
    contentPane.setBorder(
        BorderFactory.createEmptyBorder(
            30, 30, 10, 30));

    // on positionne les composants dans "contentPane"
    contentPane.add(jButton, BorderLayout.CENTER);
    contentPane.add(jLabel, BorderLayout.SOUTH);

}

}

```



Une application Swing de base (commentaires):

- **import** : les utilitaires swing sont situés dans le package `javax.swing`. Bien qu'étant une "extension standard" swing fait partie de l'API java de base livrée avec le JDK. On peut raisonnablement espérer que cette extension sera installée sur tous les sites supportant la plateforme Java 2.
- **JFrame** : Il est préférable de disposer des composants Swing dans un Container de plus haut niveau qui soit lui-même un Composant Swing. Les Containers racine (`JFrame`, `JApplet`, `JWindow`, `JDialog`) dérivent de composants AWT "lourds" (`JFrame` dérive de `Frame`) et sont conçus pour héberger les autres composants Swing qui sont des composants "poids-plume" (des composants 100% Java).
- **particularités de JFrame** : Ce container dispose de plus de services que son parent `Frame` : par exemple il est possible de spécifier l'opération à effectuer au moment de la fermeture de cette fenêtre (méthode `setDefaultCloseOperation()`).
- **ajouts de composants à un container racine (top-level)** : on ne dispose pas directement des composants à l'intérieur d'un des Containers racine. Chacun de ces Containers contient lui même un Container général (`ContentPane`) sur lequel on réalise toutes les opérations d'insertion de composants. Pour récupérer ce Container utiliser `getContentPane()`, pour affecter un Container comme container général utiliser `setContentPane()`.
- **"Look and Feel" (PLAF)** : les composants Swing sont adaptables à des aspects de divers systèmes de fenêtrage. Il est possible de donner à une application un aspect Motif sur un système Windows et réciproquement. Swing définit un "look and feel" qui lui est propre (`Metal`) et permet à l'utilisateur courageux de définir sa propre charte graphique. Dans l'exemple les modifications sur le "look and feel" sont mises en commentaire puisque l'on cherche simplement à obtenir l'aspect standard de la plateforme locale.
- **nouveaux services sur les composants** : affectation d'accélérateurs clavier (méthode `setMnemonic()` de `AbstractButton`), services généraux de `JComponent` (`setBorder()`).

La classe JComponent

Tous les composants Swing (hors les composants “racine”) dérivent de JComponent et en implantent les services :

- **bordures :**
En utilisant la méthode `setBorder()` on peut spécifier une bordure autour du composant courant. Cette Bordure peut être un espace vide (l’usage de `EmptyBorder` remplace l’utilisation de `setInsets()`) ou un dessin de bordure (implantant l’interface `Border` et rendu par la classe `BorderFactory`).
- **double buffering :**
les techniques de double-buffering permettent d’éviter les effets visuels de clignotement lors de rafraichissements fréquents de l’image du composant. On n’a plus à écrire soi-même le double-buffering, Swing gère par défaut les contextes graphiques nécessaires.
- **“bulles d’aide” (Tool tips):**
en utilisant la méthode `setToolTipText()` et en lui passant une chaîne de caractères explicative on peut fournir à l’utilisateur une petite “bulle d’aide”. Lorsque le curseur fait une pause sur le composant la chaîne explicative est affichée dans une petite fenêtre indépendante qui apparaît à proximité du composant cible.
- **utilisation du clavier :**
en utilisant la méthode `registerKeyboardAction()` on peut permettre à l’utilisateur d’utiliser uniquement le clavier pour naviguer dans l’interface utilisateur et pour déclencher des actions. La combinaison caractère + touche de modification est représentée par l’objet `KeyStroke`.
- **“pluggable look and feel” :**
au niveau global de l’application un `UIManager` gère le “look and feel”. La modification de l’aspect par `setLookAndFeel()` est soumise à des contrôles de sécurité. Derrière chaque JComponent il y a un `ComponentUI` qui gère le dessin, les événements, la taille, etc.



Approfondissements

* La mise en oeuvre des composants Swing constitue un domaine complet de spécialisation (voir notre cours SL-320)

* Outre la maîtrise des spécificités des composants standard il convient de s'intéresser à la mise en oeuvre du modèle vue-contrôleur (voir les classes `DefaultXXXModel`) et aux composants de très haut niveau : `JTree` (arbre), `JTable` (tables) ou `EditorKit` à l'intérieur d'une `JEditorPane`.



Les points importants:

Comment faire réaliser plusieurs tâches en même temps? En confiant ces tâches à des “processus” différents. Java dispose d’un mécanisme de “processus légers” (*threads*) qui s’exécutent en parallèle au sein d’une même JVM.

- La notion de *Thread*
- Création de Threads, gestion du code et des données,
- Cycle de vie d’un *Thread* et contrôles de l’exécution



Concept de thread

Qu'entend-on par thread ?

Une définition simple, mais utile, d'un ordinateur revient à dire que celui-ci possède une UC qui exécute le calcul informatique, de la mémoire morte (ROM) contenant le programme exécutant l'UC et de la mémoire vive (RAM) comprenant les données sur lesquelles le programme opère. Dans ce cas de figure, une seule tâche peut être exécutée à la fois. Une vision plus approfondie d'un ordinateur moderne admet la possibilité d'effectuer plusieurs tâches simultanément ou pour le moins donne l'impression qu'il en est ainsi.

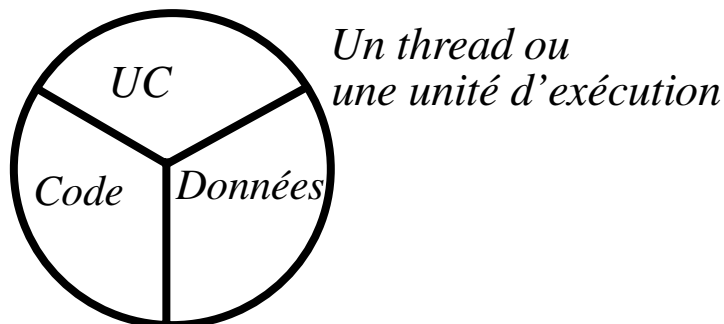
Pour l'heure, nous allons nous concentrer sur le processus de programmation plutôt que d'analyser comment l'effet est obtenu. Si nous avons au moins deux opérations réalisées, cela revient à dire que nous possédons au moins deux ordinateurs.

Nous allons considérer un *thread* ou *unité d'exécution* comme l'encastrement d'une *UC virtuelle* avec son propre code de programmation et ses propres données. La classe `java.lang.Thread` contenue dans les bibliothèques de base Java permet la création et le contrôle de nos propres threads.

Tout au long de ce module, nous employons `Thread` lorsque nous faisons référence à la classe `java.lang.Thread` et *thread* pour renvoyer à la notion d'unité d'exécution.

Concept de thread

Les trois parties d'un thread



Un thread comprend trois parties principales: L'UC, le code exécuté par cette UC et finalement les données avec lesquelles travaille le code.

Avec Java, l'UC virtuelle est incarnée dans la classe `Thread`. Lors de la mise sur pied d'un *thread*, ce dernier envoie le code à exécuter et les données à appliquer par le biais des arguments du constructeur.

Signalons que ces trois aspects sont, en fait, indépendants. Un *thread* peut exécuter le même code qu'un autre *thread* ou un code différent. Il peut, dans la même logique, accéder à des données identiques ou différentes de celles utilisées par un autre *thread*.



Création d'un Thread Java

Création d'un thread

Nous allons maintenant expliquer les phases de création d'un *thread* et examiner comment les arguments du constructeur fournissent le code et les données pour ce *thread* lors de son exécution.

Le code exécuté par le *thread*, provient de l'instance de classe dont la référence est passée en argument au constructeur du Thread.

Un constructeur Thread prend un argument qui consiste en une *instance* d'une interface Runnable. En d'autres termes, il nous faut déclarer une classe pour lancer l'interface Runnable et construire une instance de cette classe. La référence obtenue est un argument conforme au constructeur.

Par exemple:

```
public class xyz implements Runnable {
    int i;
    public void run() {
        while (true) {
            System.out.println("Hello " + i++);
        }
    }
}
```

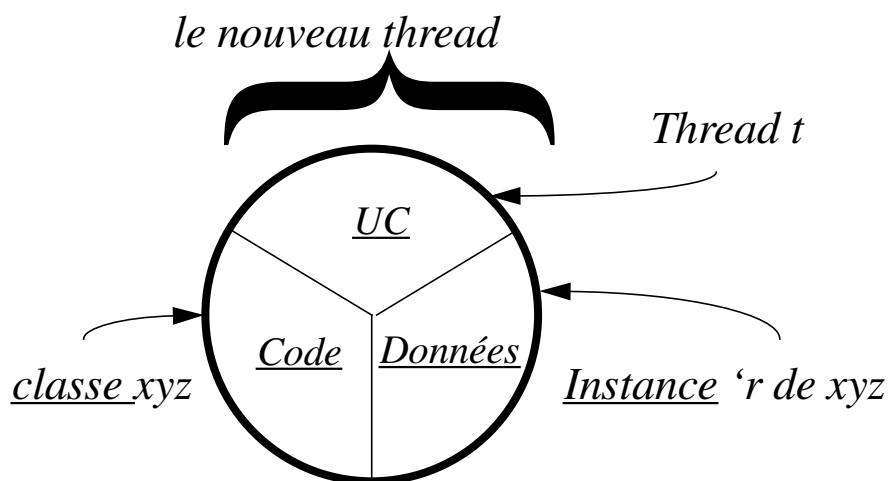
Ceci nous permet la construction d'un *thread* tel que :

```
Runnable r = new xyz();
Thread t = new Thread(r);
```


Création d'un Thread Java

Création d'un thread (suite)

A ce stade, nous avons un nouveau *thread* incarné dans la référence à Thread *t*, qui a pour but d'exécuter le code de démarrage à l'aide de la méthode `run()` de la classe *xyz*. (L'interface `Runnable` nécessite qu'une méthode `public void run()` soit disponible.) Les données utilisées par ce *thread* sont fournies par l'instance de la classe *xyz*, que nous appelons *r*.



En résumé, un *thread* est subordonné à une instance d'un objet Thread. Le code, exécuté par le thread, provient de la *classe* de l'argument envoyé au constructeur Thread. Cette classe doit mettre en oeuvre l'interface `Runnable`. Les données, utilisées par le thread, proviennent de l'*instance* précise de `Runnable` passée au constructeur de Thread.



Demarrage d'un Thread Java

Pour mettre en route le thread

Une fois créé, le *thread* n'est pas opérationnel. Pour le mettre en service, il est nécessaire d'appliquer la méthode `start()`, qui se trouve dans la classe `Thread`, c'est-à-dire qu'il faut donner l'instruction:

```
t.start();
```

Une fois cette étape franchie, l'UC virtuelle représentée dans le *thread* devient exécutable. Gardez ceci à l'esprit lorsque vous activez l'UC virtuelle.

Eligibilité d'un thread

Le fait que le *thread* soit devenu exécutable ne signifie pas nécessairement qu'il démarre instantanément. Sur une machine qui ne possède qu'une UC, il est évident que seule peut être exécutée une tâche à la fois. Nous allons maintenant examiner plus en détail la façon dont une UC est affectée lorsqu'il s'agit d'exécuter plusieurs *threads*.

Avec Java, les threads sont d'ordinaire *préemptifs* sans pour autant être soumis à un partage du temps *-time-slicing-*. (Il y a une confusion généralisée entre le terme préemptif et l'expression *time-slicing* qui, en réalité, sont deux choses bien distinctes.)

La règle régissant les priorités est que beaucoup de threads sont prêts à être exécutés, mais que seul l'un d'entre eux est effectivement lancé. Ce processus se poursuit tant qu'il est exécutable ou qu'un autre processus de priorité supérieure devient exécutable. Dans ce dernier cas de figure, le *thread* de moindre priorité est devancé par le *thread* de priorité supérieure.

Demarrage d'un Thread Java

Eligibilité d'un thread (suite)

L'interruption d'un *thread* peut être imputable à diverses raisons, dont l'appel `Thread.sleep()` lancé délibérément afin d'établir une pause ou l'attente d'une entrée/sortie sur un périphérique, fonctionnant lentement.

Tous les *threads* exécutables hors service sont conservés dans des files d'attente selon leur priorité. C'est le premier *thread* de la file d'attente de priorité supérieure qui est d'abord lancé. Lorsqu'un *thread* s'arrête pour des raisons de préemption, son état actuel est conservé et il vient s'ajouter à la file d'attente en se plaçant à la fin. De même, un *thread* qui redevient exécutable après avoir été bloqué (en sommeil ou en attente pour l'E/S par exemple) rejoint toujours la fin de la file d'attente.

Compte tenu que les *threads* de Java ne sont pas forcément soumis au time-slicing, vous devez veiller à ce que le code permette le lancement d'autres threads. Pour ce faire, vous pouvez provoquer l'appel de `sleep()`.

```
public class xyz implements Runnable {
    public void run() {
        while (true) {
            // on fait des choses
            :
            // laisser leur chance aux autres
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                // interruption! r
            }
        }
    }
}
```



Demarrage d'un Thread Java

Eligibilité d'un thread (suite)

Observez l'utilisation des méthodes `try` et `catch`. Par ailleurs, l'appel `sleep()` correspond à une méthode `static` dans la classe `Thread` et est nommée **`Thread.sleep(x)`**. L'argument précise le nombre minimum de millièmes de seconde au cours desquelles le thread doit être inactif. L'exécution du *thread* ne reprend qu'après cette période.

Une autre méthode issue de la classe `Thread`, est **`yield()`** elle laisse s'exécuter d'autres *threads*. Si d'autres threads de même priorité sont exécutables, `yield()` renvoie le *thread* demandeur à la fin de la file d'attente exécutable et laisse ainsi la voie libre pour le lancement d'un autre *thread*. La méthode `yield()` n'est effective qu'à condition qu'il existe d'autres *threads* exécutables présentant la même priorité.

Remarquez que la méthode `sleep()` permet le lancement de threads de moindre priorité alors que la méthode `yield()` n'a d'incidence que sur les threads de même priorité.

Contrôle de base des threads

Pour terminer un thread

Lorsqu'un thread a atteint la fin de la méthode `run()`, il est neutralisé. En clair, l'instance courante *ne peut plus* être exécuté.

La méthode `stop()` qui permettait d'arrêter un Thread est considérée comme très dangereuse si elle n'est pas appelée par le Thread courant, pour cette raison elle a été rendue obsolète par la version 2 de Java.

Test d'un thread

Il arrive parfois que le *thread* se trouve dans un état inconnu (ceci se produit si votre code ne contrôle pas directement un thread particulier). Au moyen de la méthode `isAlive()`, vous êtes en mesure de vous renseigner sur la viabilité d'un thread. Cette méthode n'indique pas si le thread est en cours d'exécution, mais signale qu'il a démarré et qu'un terme n'a pas été mis à son exécution.

Pour mettre les threads en attente

La méthode `join()` fait en sorte que le thread en cours attende que le thread sur lequel la méthode en question est appelée se termine. Par exemple :

```
TimerThread tt = new TimerThread (100);
tt.start ();
...
public void timeout() {
    //on attend tranquillement
    tt.join ();
    ...
    // et ça repart!
    ...
}
```

Cette méthode peut aussi être invoquée avec une temporisation en millièmes de seconde :

```
void join (long timeout);
```



D'autres façons de créer des threads

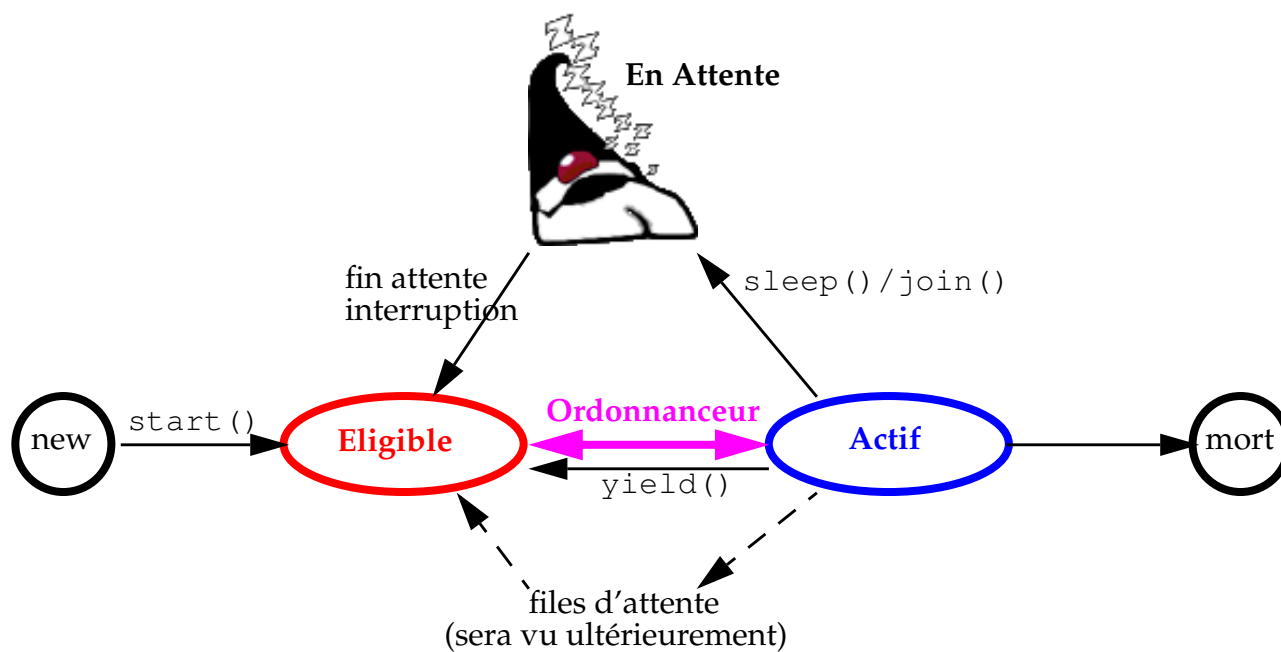
Nous avons présenté la création de threads au moyen de différentes classes réalisant l'interface Runnable. Il existe une autre approche qui consiste à définir un *thread* par dérivation de la classe Thread.

```
public class myThread extends Thread {
    public void run() {
        while (running) {
            // on en fait des choses!
            sleep(100);
        }
    }

    public static void main(String args[]) {
        Thread t = new myThread();
        t.start();
    }
}
```

Dans ce cas de figure, il n'y a qu'une seule classe, en l'occurrence myThread. Lorsque l'objet Thread est créé, aucun argument n'est fourni. Ce genre de constructeur crée un thread qui utilise sa propre structure avec une méthode run() redéfinie.

Etats d'un Thread (résumé)





Approfondissements

* Avant la version 2 de Java, les méthodes `suspend()`, `resume()`, `stop()` permettaient de suspendre l'activité d'un Thread, de le réactiver et de l'arrêter. Ces méthodes conduisant à des situations dangereuses elles ont été supprimées de l'API (voir chapitre suivant).

* On peut agir sur la priorité d'un Thread (méthode `setPriority()`) et les niveaux vont de 1 (`MIN_PRIORITY`) à 10 (`MAX_PRIORITY`). Par ailleurs on peut déclarer un Thread comme tâche de fond (méthode `setDaemon()`); une JVM s'arrête lorsque tous les Threads actifs ne sont plus que du type "daemon".

* `java.lang.ThreadGroup` permet de créer des groupes de Threads et des frontières déterminant les droits relatifs de modifications entre Threads (voir `Thread.checkAccess()`). La méthode `uncaughtException()` permet de déterminer un traitement par défaut pour les exceptions susceptibles de terminer un Thread.

* `java.lang.ThreadLocal` et `InheritableThreadLocal` permettent de gérer des variables globales à l'intérieur d'un Thread donné. On peut ainsi gérer des identifiants de session, d'utilisateur, de transaction,... qui sont accessibles comme des variables statiques mais qui sont propre au Thread courant.



Points Essentiels

Dans la mesure où l'on ne contrôle pas complètement l'ordonnement des tâches qui agissent en parallèle, il peut s'avérer nécessaire de contrôler l'accès à un objet de manière à ce que deux Threads ne le modifient pas de manière inconsistante. Pour adresser ce problème Java a adopté un modèle de "moniteur" (C.A.R. Hoare).

- Lorsqu'un Thread rentre dans un bloc de code "synchronized" rattaché à un objet, il y a un seul de ces Thread qui peut être actif, les autres attendent que le Thread qui possède le verrou ainsi acquis le restitue.
- Un Thread en attente d'une ressource peut se mettre dans une file d'attente où il attendra une notification d'un autre Thread qui rend la ressource disponible (mécanisme wait/notify).



Utilisation du mot-clé *synchronized*

Introduction

Nous allons expliquer ici comment se servir du mot-clé `synchronized` qui dote le langage Java d'un mécanisme permettant au programmeur d'avoir une emprise sur les threads qui partagent des données.

Problème

Supposons une classe représentant une pile. Au premier abord, cette classe devrait se présenter comme suit :

```
class Stack
{
    int idx = 0;
    char [] data = new char[6];

    public void push(char c) {
        data[idx] = c;
        idx++;
    }

    public char pop() {
        idx--;
        return data[idx];
    }
}
```

Notez que la classe ne tente pas de s'occuper du débordement ou du manque de pile et que la capacité de la pile est plutôt limitée. Mais laissons ceci pour nous concentrer sur d'autres aspects de la pile.

Notez que le comportement de ce modèle requiert que la valeur indicielle définisse l'indice de tableau de la prochaine cellule *vide* dans la pile, au moyen de l'approche "prédécrément, post-increment".

Utilisation du mot-clé *synchronized*

Problème (suite)

Admettons maintenant que deux *threads* font référence à une instance unique de cette classe. L'un pousse les données vers la pile et l'autre, d'une façon peu ou prou indépendante, extrait les éléments de la pile. Ceci devrait se traduire par un ajout et une suppression des données sans ambages. Cependant, un problème pourrait survenir.

Supposons que le *thread* "a" ajoute et que le *thread* "b" supprime des caractères. Le *thread* "a" vient de déposer un caractère, mais n'a pas encore incrémenté l'index du compteur. Désormais, ce *thread* est présélectionné. A ce stade, les données représentées dans notre objet sont incohérentes.

```
buffer |p|q|r| | | |
idx = 2      ^
```

En fait, au nom de la cohérence il faudrait soit que `idx = 3`, soit que le caractère n'ait pas encore été ajouté.

Dans l'hypothèse selon laquelle le *thread* "a" reprend son activité, le problème ne se pose plus; en revanche, si le *thread* "b" attend de pouvoir supprimer un caractère, il va se produire une incohérence car, alors que le *thread* "a" attend une autre occasion de se remettre au travail, le *thread* "b" saute sur l'occasion et le devance.

Nous voilà donc face à une situation chaotique où les données inscrites ne sont pas le reflet de la réalité. La méthode `pop()`, se met à décrémenter la valeur indicielle :

```
buffer |p|q|r| | | |
idx = 1      ^
```

Cette opération ignore le caractère "r" et renvoie ensuite le caractère "q". Il en résulte que la lettre "r" n'a pas été poussée, ce qui empêche de détecter le problème. Mais regardons de plus près ce qui arrive lorsque le *thread* d'origine "a" poursuit son travail.



Utilisation du mot-clé *synchronized*

Problème (suite)

Le thread "a" reprend sa tâche où il l'avait interrompue lors de l'application de la méthode `push()`. Il incrémente la valeur indicielle, ce qui nous donne :

```
buffer |p|q|r| | | |
idx = 2      ^
```

Notez que cette configuration implique que "q" est valable et que la cellule contenant "r" est la cellule vide suivante. En d'autres termes, "q" a été placé deux fois dans la pile, à l'insu de la lettre "r" qui n'y apparaît pas.

Voici un exemple illustrant les problèmes dérivés de l'accès de plusieurs threads à des données partagées. Pour prévenir tout problème, il faut un mécanisme protégeant les données contre ce genre d'accès imprévu.

Une des approches serait d'empêcher toute entrave à l'exécution du thread "a" jusqu'à ce que la partie fragile du code soit complétée. Cependant, cette pratique n'est concevable qu'avec des programmations de machines à un bas niveau et incompatible sur des systèmes complexes multi-utilisateurs.

Pour palier à cette incompatibilité, il existe un mécanisme, adopté par Java, qui traite les données fragiles avec grand soin.

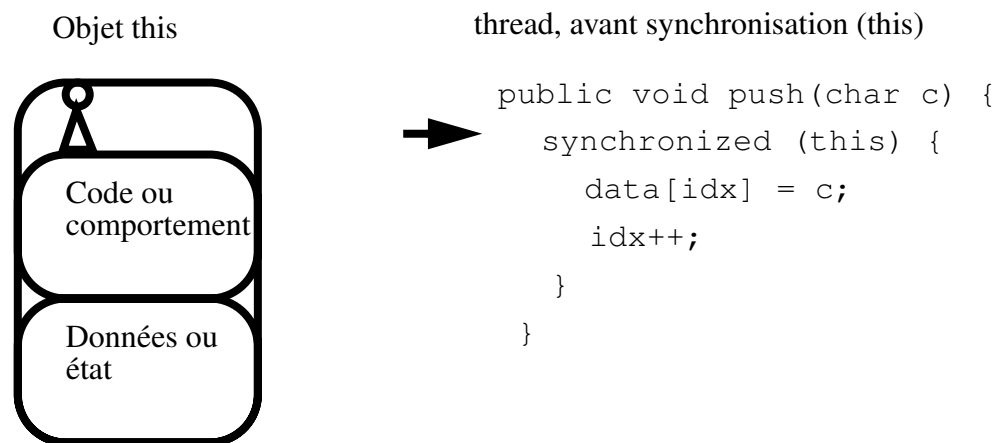
Utilisation du mot-clé `synchronized`

L'indicateur de verrouillage de l'objet

Avec Java, une instance quelconque d'un objet présente un indicateur associé qui n'est autre qu'un indicateur de verrouillage. L'interaction avec cet indicateur s'effectue moyennant un mot-clé `synchronized`. Jetons un coup d'oeil au fragment de code modifié :

```
public void push(char c) {
    synchronized(this) {
        data[idx] = c;
        idx++;
    }
}
```

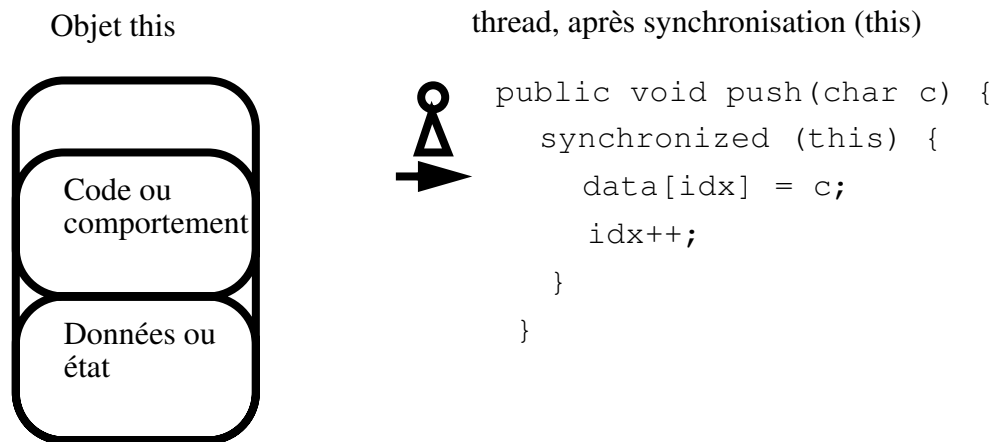
Lorsque le thread atteint l'instruction synchronisée, il tient l'objet envoyé pour l'argument et tente d'en extraire l'indicateur de verrouillage.





Utilisation du mot-clé `synchronized`

Objet indicateur de verrouillage (suite)



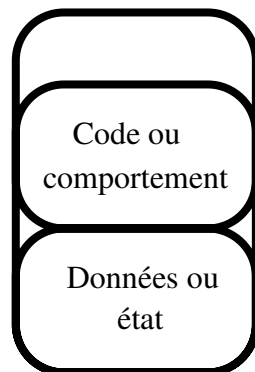
Il faut bien garder à l'esprit que l'objet `"this"` n'a pas protégé, en soi, les données qui l'identifient. Si la méthode `pop()`, sans modification, est invoquée par un autre thread, *un risque de porter atteinte à la cohérence de l'objet `"this"` n'est pas exclus*.

Utilisation du mot-clé `synchronized`

Objet indicateur de verrouillage (suite)

Pour éviter ce risque, nous devons modifier la méthode `pop` dans le même sens. En l'occurrence, nous ajoutons un appel `synchronized(this)` autour des parties fragiles de l'opération `pop()`, comme nous nous y sommes pris avec la méthode `push()`. Voici ce qui se produit si un autre thread essaie d'exécuter la méthode alors que le thread d'origine arbore l'indicateur de verrouillage :

Objet `this`
Indicateur de verrouillage absent



thread s'efforçant d'exécuter
`synchronized (this)`

```

zzz public char pop() {
      synchronized (this) {
          idx--;
          return data[idx];
      }
  }

```

Lorsque le thread tente d'exécuter l'instruction `synchronized(this)`, il essaie d'enlever l'indicateur de verrouillage de l'objet "this". L'indicateur étant absent, l'exécution avorte et le thread vient se joindre à une liste d'attente d'homologues. Cette liste est associée avec les objets indicateur de verrouillage de manière à ce que le premier thread soit lancé et puisse poursuivre son opération aussitôt que l'indicateur est restitué à l'objet.



Utilisation du mot-clé *synchronized*

Restitution de l'indicateur de verrouillage

Puisqu'un thread en attente de l'indicateur de verrouillage d'un objet ne poursuit pas son activité tant que cet indicateur n'a pas été restitué par le thread qui le détient, il va sans dire qu'il faut restituer l'indicateur lorsqu'il n'est plus nécessaire.

L'indicateur de verrouillage est restitué à son objet dès que le thread qui le détient envoie la fin du bloc associé à l'appel `synchronized()` qui l'a obtenu en premier lieu. Java prend grand soin d'assurer une restitution correcte de l'indicateur, c'est pourquoi, si le bloc synchronisé génère une exception ou si un arrêt de boucle est issu du bloc, l'indicateur sera convenablement rendu. En outre, si un thread lance deux fois l'appel synchronisé sur le même objet, l'indicateur sera libéré sans ambages du bloc le plus à l'extérieur alors que le plus à l'intérieur est, en fait, ignoré.

Ces pratiques rendent l'utilisation de blocs synchronisés beaucoup plus aisée en comparaison avec des performances semblables dans d'autres systèmes tels que les sémaphores binaires.

Utilisation du mot-clé *synchronized*

Pour assembler le tout

Comme nous l'avons déjà laissé entendre, le mécanisme `synchronized()` fonctionne uniquement à condition que le programmeur place les appels au bon endroit. Il s'agit, à présent, d'assembler une classe correctement protégée.

Observons d'abord l'accessibilité des éléments de données formant les parties fragiles de l'objet. Si elles ne sont pas privées, cela signifie qu'elles peuvent être accédées via un code sans rapport avec la définition de classe. Dans ce cas, on admet qu'aucun programmeur ne va omettre les protections de rigueur. Cette stratégie est loin d'être sûre. C'est pourquoi les données devraient toujours présenter la marque "privé".

Nous venons d'expliquer pourquoi la privacité des données revêt une importance toute spéciale. L'argument de l'instruction `synchronized()` doit, en fait, être `this`. Cette généralisation permet au langage Java d'utiliser un raccourci et au lieu d'écrire :

```
public void push(char c) {
    synchronized(this) {
        :
        :
    }
}
```

il suffit d'écrire simplement :

```
public synchronized void push(char c) {
    :
    :
}
```

qui débouche sur le même résultat.

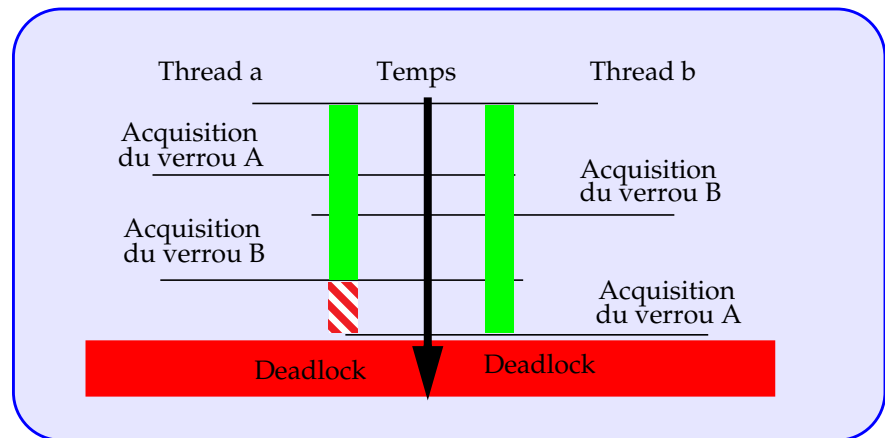


Utilisation de synchronized

Pour assembler le tout (suite)

Pourquoi choisir une méthode au détriment d'une autre ? L'utilisation de `synchronized` en tant que modificateur de méthode convertit toute la méthode en un bloc synchronisé, ce qui risque d'entraîner une durée de conservation de l'indicateur de verrouillage s'éternisant au-delà de nécessaire. D'autre part, en marquant la méthode de cette manière, les utilisateurs savent que la synchronisation a lieu et cette information peut s'avérer d'une grande utilité lors de la conception à l'encontre de deadlock, sujet traité dans la section suivante. Notez que le générateur de documentation javadoc propage le modificateur synchronisé dans les fichiers de documentation, mais l'utilisation de `synchronized(this)` n'est pas documentée.

Utilisation de synchronized



Deadlock

Dans le cadre de programmes où de nombreux threads sont en lice pour accéder aux multiples ressources, il peut se produire une situation que nous appellerons ici *deadlock* (étreinte mortelle). Cette condition intervient lorsqu'un thread attend un verrou utilisé par un autre thread qui, à son tour, attend un verrou déjà utilisé par le premier thread. Dans cette spirale infernale, aucun des threads ne peut venir à bout de son activité tant que l'autre n'a pas mis fin au processus de son bloc synchronisé. L'envoi du bloc synchronisé est condamné à échouer.

Java est incapable de détecter et encore moins de remédier à ce cercle vicieux, il incombe donc au programmeur de se prémunir contre ce genre de situation.

Voici une solution qui coule de source. Si vous avez affaire à plusieurs threads prêts à accéder à différentes ressources, déterminez l'ordre précis d'obtention desdits verrous et respectez-le au doigt et à l'oeil tout au long de la programmation.

Nous reviendrons plus en détail sur ce thème qui dépasse le simple objectif de ce cours.



Interaction de threads- wait() et notify()

Introduction

La plupart du temps, les threads sont spécialement conçus pour accomplir des tâches sans rapport entre elles. D'autres fois, en revanche, leur mission a un dénominateur commun. C'est pourquoi il convient, dans ce cas de figure, de programmer une interaction entre les threads. Pour y parvenir, vous avez le choix entre un large éventail de méthodes qui sont compatibles entre elles. Dans ce module sont présentés les mécanismes fournis dans Java.

Problème

Prenons un exemple tout simple pour expliquer les avantages qui se dégagent de l'interaction entre deux threads. Deux personnes travaillent ensemble, l'une lave la vaisselle et l'autre l'essuie. Ces deux personnes incarnent nos deux threads. Il existe entre elles un objet partagé : l'égouttoir. Admettons, en outre, qu'elles sont paresseuses et n'hésitent pas une seconde à s'asseoir si elles n'ont pas de travail à exécuter. La personne qui essuie ne peut, manifestement, pas commencer sa tâche tant que l'égouttoir ne contient pas au moins une pièce. Dans ce même ordre d'idée, si l'égouttoir est plein (celui qui lave va plus vite que son collègue), celui qui lave ne peut poursuivre tant qu'il n'y a pas un peu de place dans l'égouttoir.

Voici comment Java s'y prend pour mener à bien cette opération.

Interaction de threads- *wait()* et *notify()*

Solution

Une des solutions serait de se servir des méthodes `suspend()` et `resume()`, mais pour qu'elle fonctionne, il faut que les deux threads soient créés en coopération, puisque chacun a besoin d'un contrôle sur l'autre. Au vu de cet inconvénient, Java fournit un mécanisme de communication fondé sur les instances d'objets.

Avec Java, chaque instance d'objet possède deux files d'attente de threads associées. La première est utilisée par les threads désireux d'obtenir l'indicateur de verrouillage et est traitée dans la section "Utilisation du mot-clé `synchronized`." La seconde file sert à réaliser les mécanismes de communication `wait()` et `notify()`.

Trois méthodes sont définies dans la classe de base `java.lang.Object`, à savoir `wait()`, `notify()` et `notifyAll()`. Les méthodes `wait()` et `notify()` font l'objet de cette section. Revenons à notre exemple de vaisselle.

Le thread a lave et le thread b essuie. Tous deux ont accès à l'égouttoir. Supposons que le thread b veuille essuyer, mais que l'égouttoir soit totalement vide. Le code est alors le suivant :

```
if (drainingBoard.isEmpty())
    drainingBoard.wait();
```

Le thread b fait appel à la méthode `wait()`, ce qui entraîne l'interruption de son activité et il vient grossir la file d'attente de l'objet égouttoir. Il reste inactif tant qu'il n'est pas supprimé de la liste d'attente.

La question qui nous occupe maintenant est de savoir comment relancer le thread b. Il suffit d'appeler la méthode `notify()` comme suit :

```
drainingBoard.addItem(plate);
drainingBoard.notify();
```



Interaction de threads- wait() et notify()

Solution (suite)

A ce stade, le premier thread bloqué dans la file d'attente de l'égouttoir est supprimé de cette liste et entre en liste pour être exécuté.

Précisons que l'appel `notify()` est émis ici quel que soit l'état des threads (actif ou inactif). Cette approche ne fait pas l'unanimité, car l'appel est émis à condition que l'état vide de l'égouttoir passe à non-vide lorsque l'on y dépose une assiette. Mais il s'agit d'un détail qui excède le cadre de ce module. Ce qui nous importe, en réalité, ce sont les conditions d'utilisation de la méthode `notify()`, elle ne peut être adoptée que lorsque la file d'attente bloque des threads. Les appels de `notify()` ne sont pas mémorisés.

Par ailleurs, la méthode `notify()` libère au grand maximum le premier thread dans la file d'attente. En conséquence, si la liste d'attente comporte plusieurs threads, seul le premier est libéré. Pour libérer en bloc tous les threads, utilisez la méthode `notifyAll()`.

Par ce biais, nous pouvons coordonner nos threads qui lavent et essuient sans ambages et sans connaître leur identité. A chaque fois que nous effectuons une opération n'entravant pas le travail de l'autre thread, nous nous servons de la méthode `notify()` et l'appliquons sur l'égouttoir. D'autre part, à chaque fois que nous essayons de travailler sans pouvoir aller de l'avant parce que l'égouttoir est vide ou plein, nous adoptons la méthode `wait()` et attendons que l'égouttoir change d'état.

Interaction de threads- *wait()* et *notify()*

Mais en réalité...

Les stratégies proposées ci-dessus sont bonnes en théorie, mais la pratique se présente sous un jour plus complexe. En l'occurrence, la liste d'attente est en soi une structure de données fragiles et doit, par conséquent, être protégée à l'aide du mécanisme synchronisé. Ce qu'il nous faut retenir est qu'avant de lancer l'une des méthodes `wait()`, `notify()` ou `notifyAll()`, il est nécessaire d'obtenir l'indicateur de verrouillage pour l'objet en question. En d'autres termes, ces méthodes doivent être invoquées dans des blocs `synchronized`. Le code subit quelques modifications comme suit :

```
synchronized(drainningBoard) {
    if (drainningBoard.isEmpty())
        drainningBoard.wait();
}
```

puis :

```
synchronized(drainningBoard) {
    drainningBoard.addItem(plate);
    drainningBoard.notify();
}
```

De ce changement se dégage une remarque intéressante. Comme l'instruction `synchronized` nécessite que le thread obtienne l'indicateur de verrouillage avant d'être exécuté, ceci implique l'impossibilité pour le thread chargé de laver la vaisselle d'atteindre l'instruction `notify()` tant que le thread qui essuie la vaisselle est bloqué dans l'état `wait()`.

Dans la pratique, cela ne se produit que très rarement. En fait, l'émission de l'appel `wait()` renvoie d'abord l'indicateur de verrouillage à l'objet. Nonobstant, afin d'éviter toute déconvenue, l'appel `notify()` rend le thread inactif et le déplace tout simplement de la file d'attente vers la liste de l'indicateur de verrouillage. Dans ce cas, il ne peut poursuivre son activité tant qu'il n'a pas *recupéré* l'indicateur de verrouillage.



Interaction de threads- wait() et notify()

Mais en réalité... (suite)

Un autre aspect à prendre en compte lors de la mise en service est que la méthode `wait()` peut être arrêtée par les méthodes `notify()` `interrupt()` sur la classe `Thread`. Le cas échéant, la méthode `wait()` envoie un avertissement `InterruptedException`, qui est censé être placé dans une construction `try/catch`.

Pour assembler le tout

Prenons maintenant un exemple qui soit le reflet de la réalité. Nous continuons avec l'idée de lavage et d'essuyage, mais les caractères envoyés sur une pile supplanteront les assiettes déposées dans l'égouttoir.

Dans le domaine de l'informatique, l'exemple le plus typique met en scène un rapport entre producteur et consommateur.

Tout d'abord, nous allons nous pencher sur la présentation de la pile, puis observer avec attention les threads producteurs et consommateurs. Finalement, nous analyserons la pile et les mécanismes visant à la protéger et à mettre en route la communication via les threads.

La classe pile, appelée `SyncStack` afin de la distinguer de la classe de base du nom de `java.util.Stack`, offre l'API publique suivante :

```
public void push(char c);  
public char pop();
```



Pour assembler le tout

Producteur

Le thread producteur exécute la méthode suivante :

```
public void run() {
    char c;
    for (int i = 0; i < 20; i++) {
        c = (char) (Math.random() * 26 + 'A');
        theStack.push(c);
        System.out.println("Produced: " + c);
        try {
            Thread.sleep((int) (Math.random() * 100));
        } catch (InterruptedException e) {
            // ignore it..
        }
    }
}
```

Ceci génère 20 majuscules quelconques et les pousse sur la pile avec un délai aléatoire entre chaque opération. Le délai est compris dans une fourchette de 0 -> 100 millièmes de seconde. Chaque caractère poussé est reporté sur la console.

Pour assembler le tout

Consommateur

Le thread consommateur exécute la méthode décrite ci-dessous :

```
public void run() {
    char c;
    for (int i = 0; i < 20; i++) {
        c = theStack.pop();
        System.out.println(
            "Consumed: "+ c);
        try {
Thread.sleep((int) (Math.random() * 1000));
        } catch (InterruptedException e) {
            // ignore it..
        }
    }
}
```

Par ce biais, 20 caractères sont recueillis à partir de la pile, accusant un retard entre 0 et 2 secondes entre chaque essai. Il en découle une lenteur plus marquée lors du processus de vidage en comparaison avec le remplissage.

Nous allons à présent nous pencher sur la construction de la classe pile. Il nous faut un index et une table tampon qui ne devrait pas être trop grande, parce que le but de cet exercice est d'avoir une opération et une synchronisation correctes lors du remplissage. Nous allons prendre une table de 6 caractères.



Pour assembler le tout

Classe SyncStack

Une classe `SyncStack` nouvellement créée est normalement vide, ce qui peut aisément être arrangé au moyen de l'initialisation par défaut des valeurs, mais qui sera détaillé de façon explicite pour être plus clair. Nous pouvons entamer la construction de notre classe :

```
class SyncStack
{
    private int index = 0;
    private char [] buffer = new char[6];

    public synchronized char pop() {
    }

    public synchronized void push(char c) {
    }
}
```

Vous aurez remarqué l'absence de tout constructeur. Il serait peut-être préférable d'inclure `this` pour des raisons de style, mais, brièveté oblige, nous nous sommes permis de l'omettre.

Pour assembler le tout

Classe *SyncStack* (suite)

Analyse de push () et pop ()

Au tour des méthodes `push` et `pop` d'être au coeur de notre analyse. Nous devons appliquer le mécanisme `synchronized` pour protéger les données fragiles telles que les éléments tampon et l'index. Par ailleurs, nous devons appeler `wait()` si la méthode ne fonctionne pas, puis `notify()` pendant la réalisation de notre travail. Voici l'aspect que revêt la méthode `push()` :

```
public synchronized void push(char c) {
    while (index == buffer.length) {
        try {
            this.wait();
        } catch (InterruptedException e) {
            // ignore it..
        }
    }
    this.notify();
    buffer[index] = c;
    index++;
}
```

Notez que l'appel `wait()` est en fait `this.wait()`. La redondance n'est pas vaine, elle permet de veiller à ce que le rendez-vous ait bien lieu sur l'objet `this`. L'appel `wait()` est placé sur une construction `try/catch`. compte tenu de la suspension de l'appel `wait()` par la méthode `interrupt()`, il faut boucler notre texte au cas où le thread serait activé prématurément par l'appel `wait()`.

Examinons maintenant l'appel `notify()` qui est en réalité `this.notify()`. Malgré la redondance, il a le mérite d'être univoque. L'appel `notify()` est lancé avant l'avènement de la modification. Malgré les apparences, il ne s'agit pas d'une erreur. Tout thread en état de sommeil ne peut poursuivre son activité tant qu'il n'est pas sorti du bloc synchronisé. Nous pouvons, par conséquent, émettre l'appel `notify()` aussitôt que nous savons que nous allons pouvoir aller de l'avant avec les changements que cela implique.



Pour assembler le tout

Classe `SyncStack` (suite)

Il ne nous reste plus qu'à voir comment il faut gérer les erreurs. Vous aurez remarqué qu'il n'existe pas de code explicite pour éviter le débordement. Ceci n'est pas nécessaire, puisque seule la méthode "this", qui entre dans l'état `wait()`, permet d'insérer un élément dans notre pile et provoquer un débordement. Il est donc inutile de détecter des erreurs. Par ailleurs, nous pouvons vouer une confiance aveugle à la méthode "this" dans le cadre d'un système d'exploitation. Et pour cause, si notre logique s'avère défectueuse, nous allons nous retrouver en manque d'accès tables en dehors des limites autorisées, ce qui entraînera sur le champ l'avertissement `Exception` et l'erreur ne passera pas inaperçue. Pour couvrir d'autres cas, vous pouvez utiliser l'exception `Runtime` pour placer vos propres contrôles.

Il en va de même pour la méthode `pop()` :

```
public synchronized char pop() {
    while (index == 0) {
        try {
            this.wait();
        } catch (InterruptedException e) {
            // ignore it..
        }
    }
    this.notify();
    index--;
    return buffer[index];
}
```

Il ne nous reste plus qu'à mettre ces fragments dans des classes complètes et ajouter un cadre pour les rendre exécutables. Le code final est illustré dans les pages suivantes.

Classe SyncStack

SyncTest.java

```
package mod13;
public class SyncTest
{
    public static void main(String args[]) {
        SyncStack stack = new SyncStack();
        Runnable source = new Producer(stack);
        Runnable sink = new Consumer(stack);
        Thread t1 = new Thread(source);
        Thread t2 = new Thread(sink);
        t1.start();
        t2.start();
    }
}
```

Producer.java

```
package mod13;
public class Producer implements Runnable
{
    SyncStack theStack;

    public Producer(SyncStack s) {
        theStack = s;
    }

    public void run() {
        char c;
        for (int i = 0; i < 20; i++) {
            c = (char) (Math.random() * 26 + 'A');
            theStack.push(c);
            System.out.println("Produced: " + c);
            try {
                Thread.sleep((int) (Math.random() * 100));
            } catch (InterruptedException e) {
                // ignore it..
            }
        }
    }
}
```



Classe *SyncStack*

Consumer.java

```
package mod13;
public class Consumer implements Runnable
{
    SyncStack theStack;

    public Consumer(SyncStack s) {
        theStack = s;
    }

    public void run() {
        char c;
        for (int i = 0; i < 20; i++) {
            c = theStack.pop();
            System.out.println("Consumed: " + c);
            try {
                Thread.sleep((int) (Math.random() * 1000));
            } catch (InterruptedException e) {
                // ignore it..
            }
        }
    }
}
```


Classe SyncStack

SyncStack.java

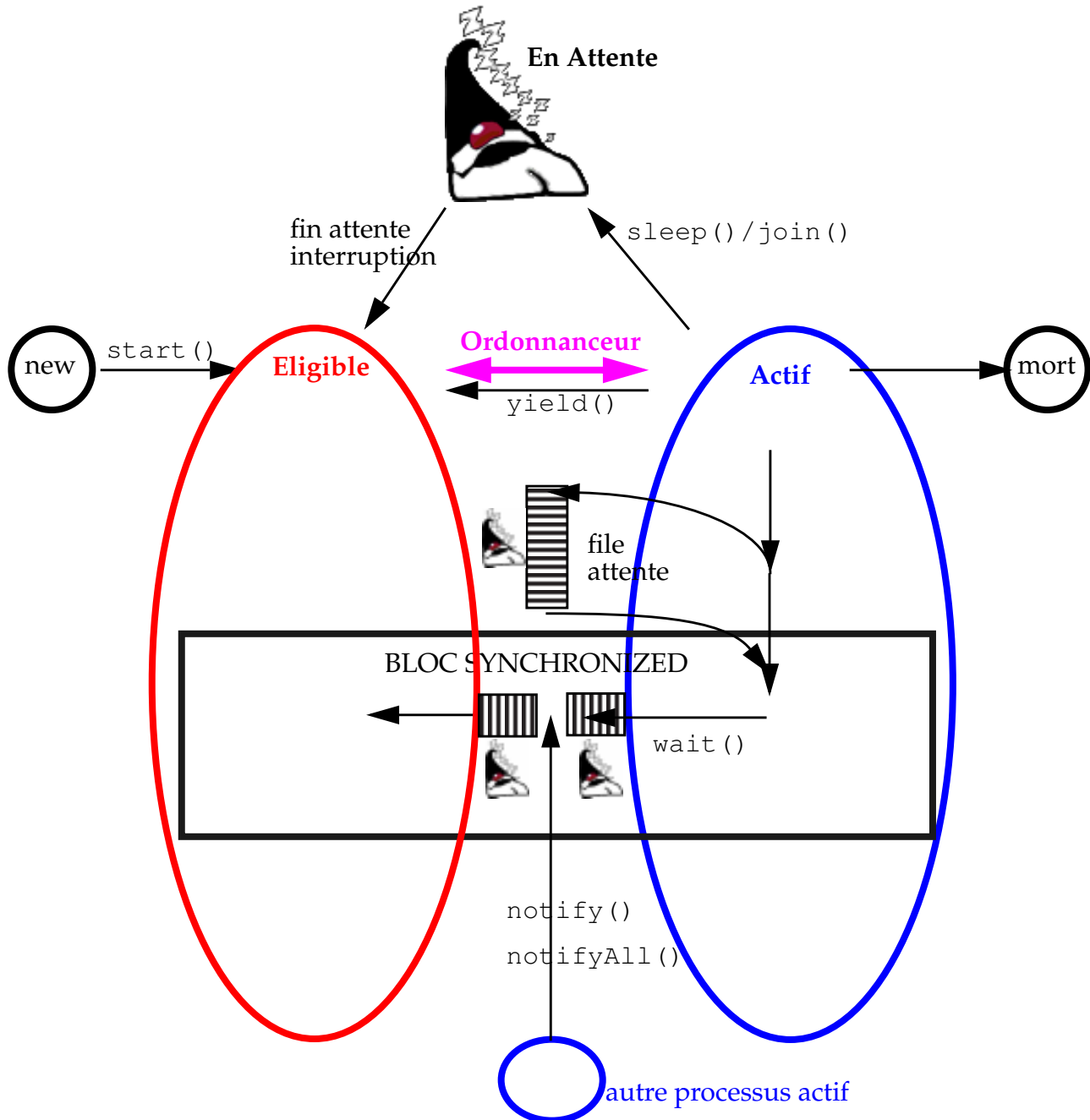
```
package mod13;
public class SyncStack
{
    private int index = 0;
    private char [] buffer = new char[6];

    public synchronized char pop() {
        while (index == 0) {
            try {
                this.wait();
            } catch (InterruptedException e) {
                // ignore it..
            }
        }
        this.notify();
        index--;
        return buffer[index];
    }

    public synchronized void push(char c) {
        while (index == buffer.length) {
            try {
                this.wait();
            } catch (InterruptedException e) {
                // ignore it..
            }
        }
        this.notify();
        buffer[index] = c;
        index++;
    }
}
```



Etats d'un Thread (résumé)



Remarques : 1) Dans un bloc synchronized le processus peut être actif ou préempté par l'ordonnanceur. 2) Lorsque un processus reçoit un notify() il n'est pas forcément réactivé immédiatement: il peut avoir à attendre qu'un autre processus relache le verrou sur l'objet synchronisé courant.

Approfondissements

** Les méthodes de thread `suspend()` et `resume()` étant obsolètes on peut chercher un moyen pour permettre à un thread de répondre à une demande de suspension.*

```
public class ControlledThread extends Thread {
    static final int SUSP = 1;
    static final int STOP = 2;
    static final int RUN = 0;
    private int state = RUN;

    public synchronized void setState(int s) {
        state = s;
        if ( s == RUN ) { //corriger si s==STOP et state==SUSP
            notify();
        }
    }

    public synchronized boolean checkState() {
        while ( state == SUSP ) {
            try {
                wait();
            } catch (InterruptedException e) {
                // ignore
            }
        }
        if ( state == STOP ) {
            return false;
        }
        return true;
    }

    public void run() {
        while ( true ) {
            doSomething();

            // Be sure shared data is in consistent state in
            // case the thread is waited or marked for exiting
            // from run()
            if ( !checkState() ) {
                break;
            }
        }
    }
}
```



Points essentiels :

Java offre une vision des entrée/sorties portables basées sur la notion de flot (*Stream*).

- Il existe deux catégories de flots de base : les flots d'octets (InputStream, OutputStream) et les flots de caractères (Reader, Writer).
- Certains flots sont associés à des ressources qui fournissent des données (fichiers, tableaux en mémoire, lignes de communications,...)
- D'autres types de flots transforment la manière dont on opère sur les données en transit (E/S bufferisée, traduction de données, etc.)
- C'est en combinant les services de ces différents types de flot que l'on contrôle les opérations d'entrées/sorties.



Fots E/S avec Java

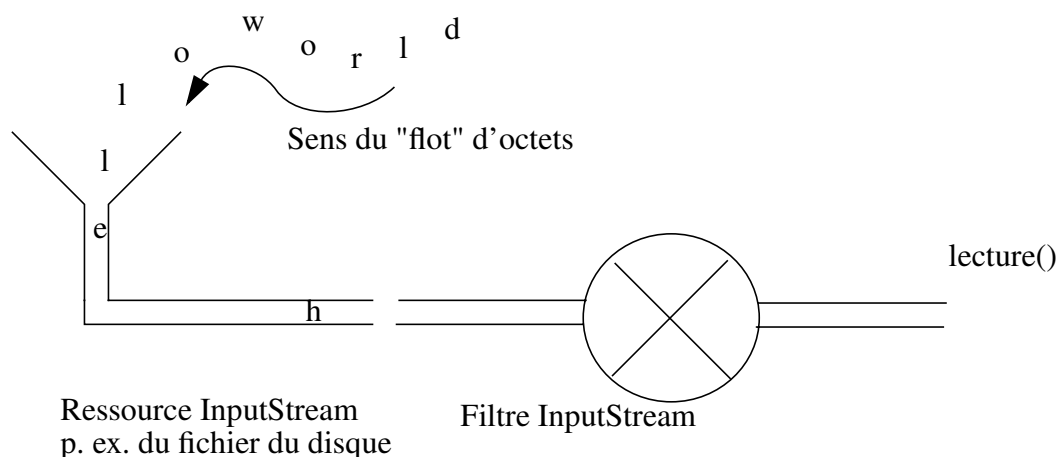
Dans ce module, nous allons examiner comment le langage Java gère les Entrées/Sorties (y compris sur `stdin`, `stdout` et `stderr`) par le biais de flots (*Stream*). Par la suite, nous approfondirons le maniement des fichiers et des données qui s'y trouvent.

Les fondements de la notion de flot

Un flot est soit une source d'octets, soit une destination pour les octets. L'ordre est important. Par exemple, un programme souhaitant lire à partir d'un clavier peut se servir d'un flot pour mener à bien cette action.

Il existe deux catégories fondamentales de flots, à savoir les flots d'entrée, à partir desquels on peut lire et les flots de sortie qui, au contraire, acceptent l'écriture mais ne peuvent être lus.

Dans le package `java.io`, certains flots ont pour origine une ressource, ils peuvent lire ou écrire dans ressource déterminée telle qu'un fichier ou une zone mémoire. D'autres flots sont appelés **filtres**. Un filtre d'entrée est créé moyennant une connexion à un flot d'entrée existant, de sorte que, lorsque vous tentez de lire à partir du filtre d'entrée, vous obtenez les données extraites, à l'origine, sur cet autre flot d'entrée.



Flots E/S avec Java

Méthodes *InputStream*

```
int read()
int read(byte[])
int read(byte[], int, int)
```

Ces trois méthodes fournissent des octets. La méthode `read` renvoie un argument `int` contenant un octet lu à partir du flot ou la valeur `-1` indiquant la fin de fichier. Les deux méthodes restantes remplissent une table d'octets avec des octets lus et en renvoient le nombre. Les deux arguments `int` de la troisième méthode indiquent un sous-ensemble de la table cible.



Pour profiter d'une efficacité optimale, lisez toujours les données par blocs les plus grands possible.

```
void close()
```

Il est recommandé de fermer un flot lorsque vous avez fini de l'utiliser.

```
int available()
```

Cette instruction signale le nombre d'octets prêts à être lus dans le flot. Une opération de lecture réelle succédant à cet appel peut éventuellement renvoyer plus d'octets.

```
skip(long)
```

Cette méthode permet de "sauter" un nombre déterminé d'octets provenant du flot.



Flots E/S avec Java

Méthodes *InputStream* (suite)

```
markSupported()  
mark(int)  
reset()
```

Ces méthodes visent à effectuer des opérations de "rejet" sur un flot à condition que celui-ci les prenne en charge. La méthode `markSupported()` renvoie `true` si les méthodes `mark()` et `reset()` sont opérationnelles dans le cadre de ce flot spécifique. La méthode `mark(int)` sert à indiquer que le point courant dans le flot devrait être noté et qu'il faudrait affecter une quantité suffisante de mémoire tampon, dans le but d'admettre au moins le nombre de caractères de l'argument donné. A la suite d'opérations `read()` successives, la méthode `reset()` tend à repositionner le flot d'entrée sur le premier point mémorisé.

Flots E/S avec Java

Méthodes `OutputStream`

```
write(int)
write(byte[])
write(byte[], int, int)
```

Ces méthodes écrivent dans le flot de sortie. A l'instar des flots d'entrée, il est recommandé d'écrire les données par blocs les plus grands possibles.

```
close()
```

Il est préférable de fermer les flots de sortie après avoir terminé de les utiliser.

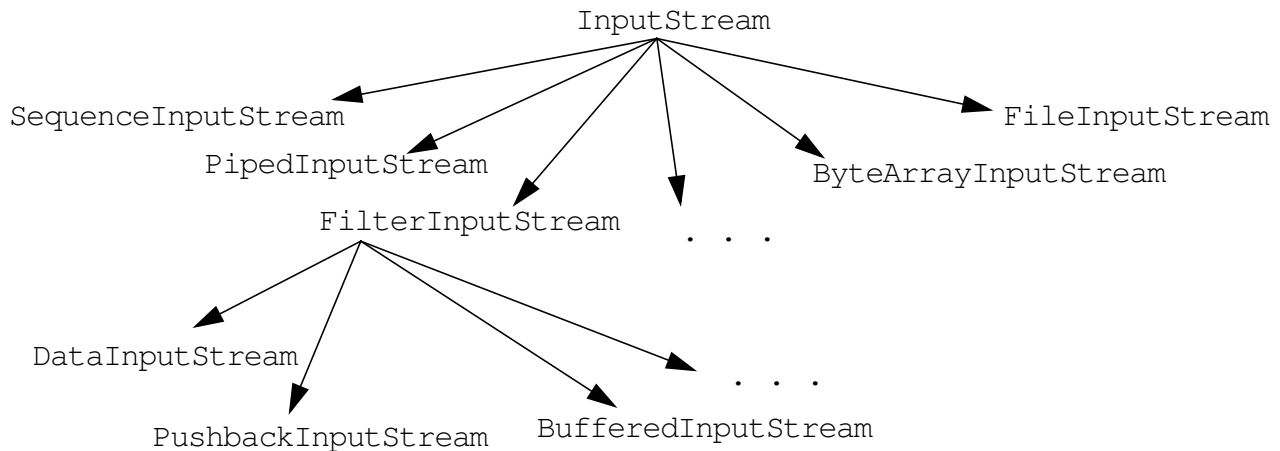
```
flush()
```

Il arrive parfois que le flot de sortie tamponne les écritures avant de les écrire réellement. La méthode `flush()` vous donne les moyens de purger les buffers..



Streams de base

De manière simplifiée voici comment se présente la hiérarchie au sein du package `java.io`



FileInputStream et FileOutputStream

Il s'agit de classes mettant en jeu des flots issus de ressources et, comme leur nom l'indique, elles utilisent des fichiers disque. Les constructeurs de ces classes vous permettent de préciser le chemin d'accès du fichier auquel elles sont connectées. La construction d'une classe `FileInputStream` est conditionnée par l'existence du fichier associé et par son accès en lecture. Lors de la construction d'une classe `FileOutputStream`, le fichier de sortie, s'il existe encore, est écrasé.

```
FileInputStream infile =
    new FileInputStream("myfile.dat");

FileOutputStream outfile =
    new FileOutputStream("results.dat");
```

Classes Stream de base

BufferedInputStream et BufferedOutputStream

Ces filtres permettent de bufferiser les d'E/S et donc de les optimiser.

DataInputStream et DataOutputStream

Ces filtres permettent la lecture et l'écriture de types de base Java au moyen de flots. Voici une palette de méthodes spéciales pour les types de base :

Méthodes DataInputStream

```
byte readByte()  
long readLong()  
double readDouble()
```

Méthodes DataOutputStream

```
void writeByte(byte)  
void writeLong(long)  
void writeDouble(double)
```

Notez que les méthodes de `DataInputStream` vont de pair avec les méthodes de `DataOutputStream`.

PipedInputStream et PipedOutputStream

Les "tubes" de communication (piped streams) servent de pont de communication entre *threads*. Un objet `PipedInputStream` dans un *thread* reçoit ses données en entrée à partir d'un objet `PipedOutputStream` complémentaire situé dans un autre *thread*. .



Flots d'entrée sur URL

En sus de l'accès de fichiers de base, Java est munie d'URL (Uniform Resource Locators) destinées à accéder à des objets via un réseau. Vous utilisez un objet URL lorsque vous accédez aux sons et images à l'aide de la méthode `getDocumentBase()` pour des applets :

```
String imageFile = new String ("images/Duke/T1.gif");
images[0] = getImage(getDocumentBase(), imageFile);
```

Vous pouvez toutefois fournir un URL directement comme suit :

```
java.net.URL imageSource;
try {
    imageSource = new URL("http://mysite.com/~info");
} catch (MalformedURLException e) {}
images[0] = getImage(imageSource, "Duke/T1.gif");
```

Pour ouvrir un flot d'entrée

En outre, vous pouvez ouvrir un flot d'entrée à partir d'une URL :

```
InputStream is;
String datafile = new String("Data/data.1-96");
byte buffer[] = new byte[24];
try {
    // new URL throws a MalformedURLException
    // URL.openStream() throws an IOException
    is = (new URL(getDocumentBase(), datafile)).openStream();
} catch (Exception e) {}
```

A présent, vous êtes en mesure d'utiliser `is` pour lire les informations, :

```
try {
    is.read(buffer, 0, buffer.length);
} catch (IOException e1) {}
```



N'oubliez pas que la plupart des utilisateurs ont une politique de sécurité visant à empêcher les applets d'accéder aux fichiers.

Readers et Writers

Unicode

En interne Java utilise le format Unicode, comme ces caractères sont représentés sur 16 bits on est obligé d'utiliser des flots spéciaux pour les manipuler: ce sont les `Reader` et les `Writer`.

Les classes `InputStreamReader` et `OutputStreamWriter` font le pont avec les flots d'octets en assurant de manière implicite ou explicite les conversions nécessaires.

Octet <-> Conversion de caractères

Par défaut, si vous construisez simplement un `Reader` ou un `Writer` connecté à un flot, les règles de conversion qui s'appliquent sont celles entre le codage de la plateforme locale et Unicode. Dans la plupart des pays européens utilisant les caractères "latins", l'encodage des caractères suit la norme ISO 8859-1. ("Cp1252" sous Windows).

Vous pouvez sélectionner un autre type de codage d'octets, à l'aide d'une des listes regroupant les formes de codage reconnues que vous trouverez dans la documentation pour l'outil `native2ascii`.

Pour lire une entrée à partir d'un codage de caractère non local ou même la lire à partir d'une connexion réseau avec un type différent de machine, vous pouvez construire un `InputStreamReader` par le biais d'un codage de caractères explicite comme suit :

```
ir = new InputStreamReader(flotentree, "ISO8859_1")
```



Si vous lisez les caractères d'une connexion réseau, nous vous enjoignons d'utiliser cette formulation, faute de quoi votre programme sera toujours tenté de convertir les caractères qu'il lit comme s'il s'agissait d'une représentation locale, ce qui ne serait pas le reflet de la réalité .



Readers et Writers

BufferedReader et BufferedWriter

Il est préférable d'enchaîner un `BufferedReader` ou un `BufferedWriter`, sur un `InputStreamReader` ou `OutputStreamWriter` car la conversion entre formats donne des résultats plus probants lorsqu'elle est effectuée par blocs. N'oubliez pas d'utiliser la méthode `flush()` sur un `BufferedWriter`.

Lecture de chaîne d'entrées

```
public class CharInput {
    public static void main(String args[]) {
        String s;
        InputStreamReader ir;
        BufferedReader in;
        ir = new InputStreamReader(System.in);
        in = new BufferedReader(ir);

        while ((s = in.readLine()) != null) {
            System.out.println("Read: " + s);
        }
    }
}
```

Fichiers

Avant de pouvoir appliquer E/S sur un fichier, il vous faut obtenir l'information de base concernant ce fichier. La classe `File` met à disposition une série d'utilitaires permettant de travailler avec des fichiers et d'obtenir des informations à leur sujet.

Pour créer un nouvel objet File

```
1 File myFile;
  myFile = new File("monfichier");

1 myFile = new File("repertoire", "monfichier");
  // mieux si repertoire et/ou fichiers sont des vars

1 File myDir = new File("repertoire");
  myFile = new File(myDir, "monfichier");
```

Le constructeur que vous utilisez dépend souvent d'autres objets fichier auxquels vous accédez. Par exemple, si vous utilisez un fichier dans votre application, utilisez le premier constructeur. En revanche, si vous utilisez plusieurs fichiers provenant d'un répertoire commun, il est plus pratique de faire appel au second ou troisième constructeur.

Un objet `File` peut faire office d'argument de constructeur pour des objets `FileInputStream` et `FileOutputStream`. Vous préservez ainsi une indépendance vis-à-vis de la convention locale de notation de la hiérarchie du système de fichiers.



Tests de fichiers et utilitaires

Une fois un objet `File` créé, vous pouvez librement opter pour une des méthodes suivantes afin de recueillir des information au sujet du fichier :

Noms de fichiers

- |** `String getName()`
- |** `String getPath()`
- |** `String getAbsolutePath()`
- |** `String getParent()`
- |** `boolean renameTo(File newName)`

Tests de fichiers

- |** `boolean exists()`
- |** `boolean canWrite()`
- |** `boolean canRead()`
- |** `boolean isFile()`
- |** `boolean isDirectory()`
- |** `boolean isAbsolute();`

Information générale axée sur les fichiers et utilitaires

- |** `long lastModified()`
- |** `long length()`
- |** `boolean delete()`

Utilitaires en rapport avec les répertoires

- |** `boolean mkdir()`
- |** `String[] list()`

Fichiers à accès direct

Le langage Java vous fournit une classe `RandomAccessFile` pour prendre en charge les fichiers à accès direct.

Pour créer un fichier d'accès direct

Vous avez le choix entre deux options pour ouvrir un fichier à accès direct:

I Par le nom du fichier

```
myRAFile = new RandomAccessFile(String name, String mode);
```

I Par un objet `File`

```
myRAFile = new RandomAccessFile(File file, String mode);
```

L'argument `mode` détermine si vous disposez d'un accès en lecture seule ("r") ou en lecture/écriture ("rw").

Voici comment procéder, lors de l'ouverture d'un fichier base de données, en vue d'une mise à jour :

```
RandomAccessFile myRAFile;  
myRAFile = new RandomAccessFile("stock.dbf", "rw");
```

Pour accéder à l'information

Les objets `RandomAccessFile` escomptent lire et écrire les informations comme les flots d'E/S données: vous pouvez accéder à toutes les opérations des classes `DataInputStream` et `DataOutputStream`.



Fichiers à accès direct

Pour accéder à l'information (suite)

Le langage Java propose diverses méthodes permettant de parcourir le fichier :

```
l long getFilePointer();
```

Renvoie la position courante du pointeur de fichier.

```
l void seek(long pos);
```

Fixe le pointeur de fichier sur la position absolue indiquée qui est précisée par un décalage d'octets partant du début du fichier. La position 0 marque le début du fichier.

```
l long length();
```

Renvoie la longueur du fichier. La position `length()` marque la fin du fichier.

Pour insérer des informations

:

```
myRAFile = new RandomAccessFile("java.log", "rw");  
myRAFile.seek(myRAFile.length());  
// Any subsequent write()s will be appended to the file
```

Approfondissements

* *La lecture/écriture d'objets sur les flots sera abordée dans un chapitre ultérieur (serialization)*

* *Voir les packages `java.util.zip` et `java.util.jar` pour des utilitaires de compression ou d'accès aux fichiers Jar.*

* *Voir également dans `java.io` la classe `StreamTokenizer` (analyseur de texte)*





Points essentiels :

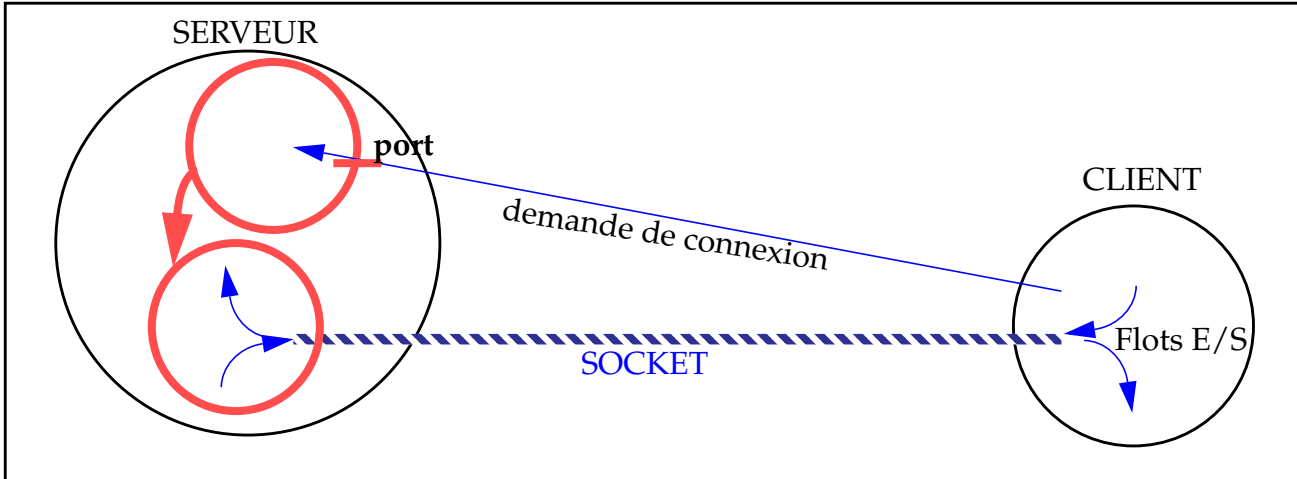
Le package `java.net` offre des services permettant des connexions directes au travers du réseau. Le paradigme de “socket” est utilisé pour établir des connexions.

- communications sur TCP (flots)
- communications via UDP (datagrammes).
- datagrammes avec diffusion multiple (multicast).

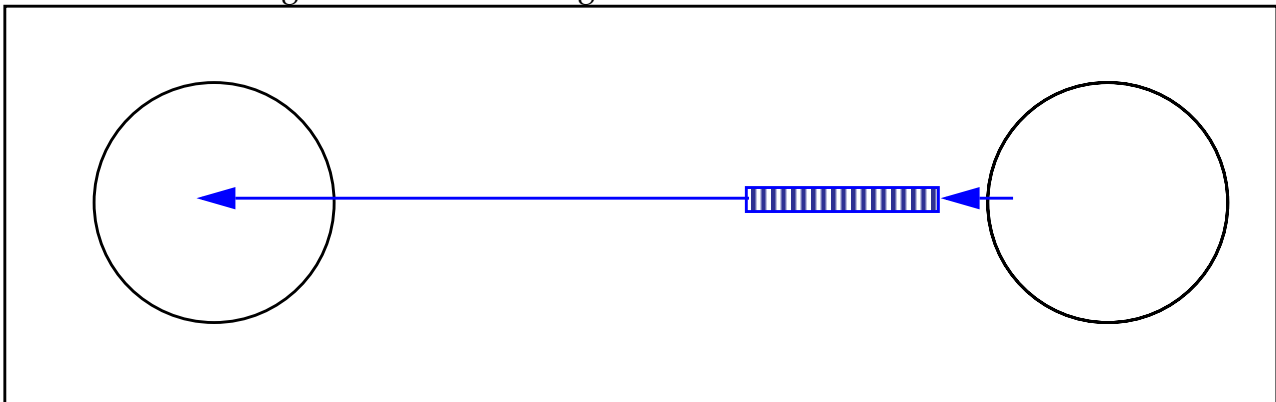


Modèles de connexions réseau en Java

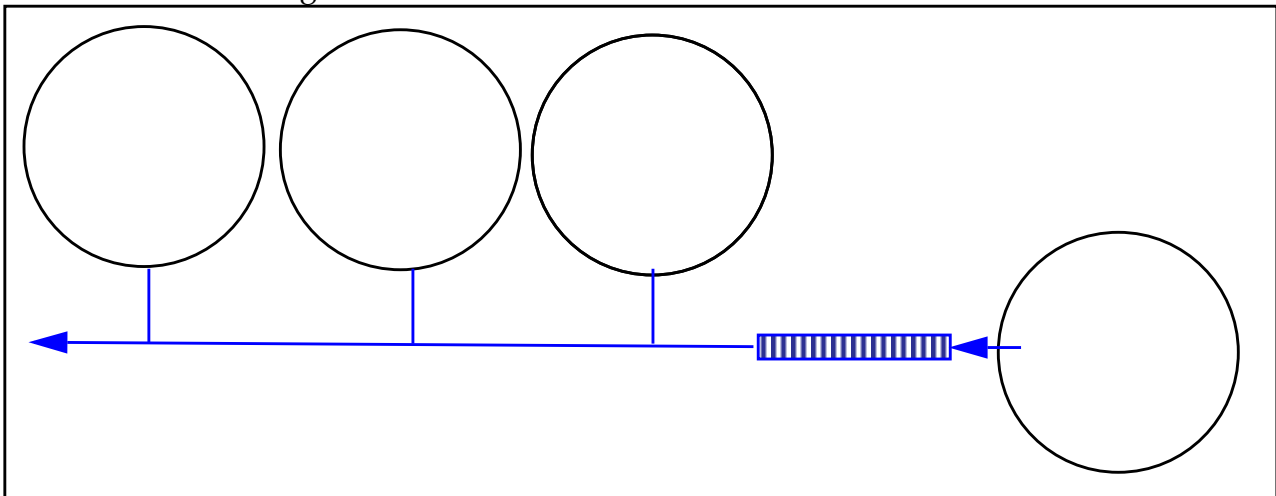
Accès programmatiques aux "sockets" (canaux de communication bidirectionnels) sur TCP/IP



Programmation de Datagrammes UDP



Datagrammes avec "MultiCast"



Programmation réseau en Java

Sockets

Les Sockets sont des points d'entrée de communication bi-directionnelle entre deux applications sur un réseau.

Les différents types de Socket conditionnent la façon dont les données vont être transférées :

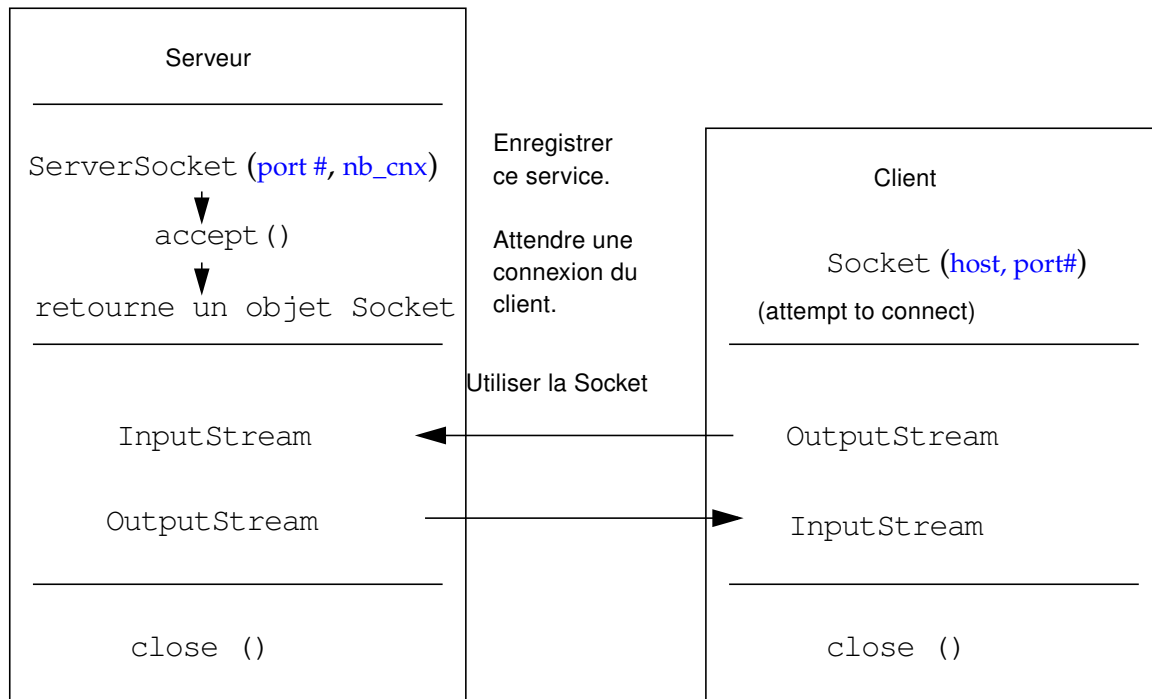
- **Stream sockets (TCP)** – Permettent d'établir une communication en mode connecté. Un flot continu est établi entre les deux correspondants : les données arrivent dans un ordre correct et sans être corrompues.
- **Datagram sockets (UDP)** – Permettent d'établir une connexion en mode non-connecté, que l'on appelle aussi mode Datagramme. Les données doivent être assemblées et envoyées sous la forme de paquets indépendants de toute connexion. Un service non connecté est généralement plus rapide qu'un service connecté, mais il est aussi moins fiable : aucune garantie ne peut être émise quand au fait que les paquets seront effectivement distribués correctement -ils peuvent être perdus, dupliqués ou distribués dans le désordre-.



Le modèle réseau de Java.

Dans Java, les sockets TCP/IP sont implantées au travers de classes du package `java.net`. Voici la façon dont ces classes sont utilisées.

:



Dans ce modèle, le fonctionnement est le suivant :

- Le Serveur enregistre son service sous un numéro de port, indiquant le nombre de client qu'il accepte de faire patienter à un instant T. Puis, le serveur se met en attente sur ce service par la méthode `accept ()` de son instance de `ServerSocket`.
- Le client peut alors établir une connexion avec le serveur en demandant la création d'une socket à destination du serveur pour le port sur lequel le service a été enregistré.
- Le serveur sort de son `accept ()` et récupère une `Socket` en communication avec le Client. Ils peuvent alors utiliser des `InputStream` et `OutputStream` pour échanger des données.

Principe d'un Serveur TCP/IP

Exemple de code de mise en oeuvre d'un serveur TCP/IP

```
import java.net.*;
import java.io.*;

public class Serveur {
    ServerSocket srv ;

    public Serveur( int port) throws IOException{
        srv = new ServerSocket(port) ;
    }

    public void go() {
        while (true) {
            try {
                Socket sck = srv.accept() ;
                dialogue(sck.getInputStream(), sck.getOutputStream());
                sck.close() ;
            } catch (IOException exc) {
            }
        }
    }

    public void dialogue(InputStream is, OutputStream os)
        throws IOException {
        DataOutputStream dos = new DataOutputStream(os) ;
        dos.writeUTF ("Hello net World") ;
    }

    public static void main (String[] args) throws Exception {
        Serveur serv = new Serveur(Integer.parseInt(args[0])) ;
        serv.go() ;
    }
}
```



Principe d'un Client TCP/IP

Le client correspondant :

```
import java.net.*;
import java.io.*;

public class Client {
    Socket sck ;

    public Client( String host, int port) throws IOException{
        sck = new Socket(host, port) ;
    }

    public void go() {
        try{
            dialogue(sck.getInputStream(), sck.getOutputStream()) ;
        } catch (IOException exc) { }
    }

    public void stop() {
        try {
            sck.close() ;
        } catch (IOException exc) { }
    }

    public void dialogue(InputStream is, OutputStream os)
        throws IOException {
        DataInputStream dis = new DataInputStream(is) ;
        System.out.println(dis.readUTF()) ;
    }

    public static void main (String[] args) throws Exception {
        Client cli = new Client( args[0],
            Integer.parseInt(args[1])) ;
        cli.go() ;
        cli.stop() ;
    }
}
```

échanges UDP

On n'a pas ici de connexion ouverte en permanence. Les paquets, autonomes, sont transférés avec leur propres informations d'adresse. Le service n'est pas "fiable" car il y a des risques de perte, ou de duplication de paquets. L'ordre d'arrivée n'est pas garanti.

Pour limiter les incidents il vaut mieux limiter la taille des paquets envoyés de manière à ce qu'ils n'occupent qu'un seul paquet IP.

Les objets fondamentaux :

- `DatagramSocket` : détermine un canal (socket) UDP. Pour un serveur on précisera le port (pas nécessaire pour le client)

```
DatagramSocket serverSock = new DatagramSocket(9789);  
DatagramSocket clientSock= new DatagramSocket() ;
```

- `DatagramPacket` : structure d'accueil des données et des informations d'adresse.
Les méthodes `getData()`, `getAddress()`, `getPort()` permettent de récupérer ces informations.

```
// pour un envoi  
sendPack = new DatagramPacket(byteBuff, len, addr, port);  
socket.send(sendPack) ;  
// pour une reception  
recvPack = new DatagramPacket(byteBuffer, len) ;  
socket.receive(recvPack) ;
```

- `InetAddress` : permet de produire une adresse inet à partir d'une désignation (méthode `getByName()`) ou de la machine locale (`getLocalHost()`)



Exemple de Serveur UDP

Ce serveur reçoit un message d'un client et le renvoie précédé d'un autre message.

On utilise les données d'adressage du paquet reçu du client pour reexpédier les données

```
import java.io.* ;
import java.net.* ;
import java.sql.* ;

public class DatagServer {
    public static final int DATA_MAX_SIZE = 512 ;
    DatagramSocket veille ;

    public DatagServer( int port) throws IOException {
        veille = new DatagramSocket(port) ;
    }

    public void go() {
        byte[] recBuffer = new byte[DATA_MAX_SIZE] ;
        while (true) {
            try {
                // ***** on ecoute
                DatagramPacket recvPack =
                    new DatagramPacket(recBuffer, recBuffer.length) ;
                veille.receive(recvPack) ;
                // ***** on prepare les streams et on dialogue
                ByteArrayInputStream biz =
                    new ByteArrayInputStream(recvPack.getData()) ;
                ByteArrayOutputStream boz =
                    new ByteArrayOutputStream() ;
                lireEcrire(biz,boz) ;
                // ***** on renvoie
                DatagramPacket sendPack =
                    new DatagramPacket(boz.toByteArray(), boz.size(),
                        recvPack.getAddress(),
                        recvPack.getPort()) ;
                veille.send(sendPack) ;
            } catch (IOException exc) { }
        }
    } // go
}
```

```
public void lireEcrire(InputStream is, OutputStream os)
    throws IOException {
    DataInputStream dis = new DataInputStream(is) ;
    String mess = dis.readUTF() ;
    System.out.println(mess) ;
    DataOutputStream dos = new DataOutputStream(os) ;
    dos.writeUTF("reçu: " + mess ) ;
}

public static void main (String[] args) throws Exception {
    DatagServer serv =
        new DatagServer(Integer.parseInt(args[0])) ;
    serv.go() ;
} //End main
}
```



Exemple de client UDP

Correspond au serveur précédent : on envoie un message et on le reçoit en echo précédé de celui du serveur.

```
import java.io.* ;
import java.net.* ;
import java.sql.* ;

public class DatagClient {
    public static final int DATA_MAX_SIZE = 512 ;

    DatagramSocket sock ;
    InetAddress inetServ ;
    int port ;

    public DatagClient( String server, int sport )
        throws IOException {
        sock = new DatagramSocket() ;
        inetServ= InetAddress.getBy_name(server) ;
        port = sport ;
    }

    public void go() {
        byte[] recBuffer = new byte[DATA_MAX_SIZE] ;
        try {
            ByteArrayOutputStream boz =
                new ByteArrayOutputStream();
            ecrire(boz) ;
            DatagramPacket sendPack =
                new DatagramPacket(boz.toByteArray(), boz.size(),
                    inetServ, port) ;
            sock.send(sendPack) ;
            // ***** on ecoute
            DatagramPacket recvpack =
                new DatagramPacket(recBuffer, recBuffer.length) ;
            sock.receive(recvpack) ;
            // ***** on prepare les streams et on dialogue
            ByteArrayInputStream biz =
                new ByteArrayInputStream(recvpack.getData()) ;
            lire(biz) ;
            // ***** on renvoie
        } catch (IOException exc) { }
    } // go
}
```

```
public void ecrire( OutputStream os)
    throws IOException {
    DataOutputStream dos = new DataOutputStream(os) ;
    dos.writeUTF("bonjour") ;
}

public void lire(InputStream is)
    throws IOException {
    DataInputStream dis = new DataInputStream(is) ;
    String mess = dis.readUTF() ;
    System.out.println(mess) ;
}

public static void main (String[] args) throws Exception {
    DatagramClient cli = new DatagramClient(args[0],
        Integer.parseInt(args[1])) ;
    cli.go() ;
} //End main
}
```



UDP en diffusion (Multicast)

Une adresse de diffusion (multicast) est une adresse comprise entre 224.0.0.0 et 239.255.255.255.

Des `MulticastSocket` permettent de diffuser des données simultanément à un groupe d'abonnés. Sur une telle socket on peut s'abonner à une adresse multicast par `joinGroup(InetAddress mcastaddr)` ou se désabonner par `leaveGroup(InetAddress)`.

Le paramètre TTL de ces sockets permet de fixer le nombre maximum de routeurs traversés - si ces routeurs le permettent- (important si on veut limiter la diffusion à l'extérieur d'une entreprise)

Exemple de Serveur Multicast

Diffuse un `java.sql.Timestamp` toute les secondes. Les paquets ne se transmettent pas au travers des relais réseaux (TTL = 1).

```

1  import java.io.* ;
2  import java.net.* ;
3  import java.sql.* ;
4
5  public class MultigServer {
6      public static final int PORT = 9999 ;
7      public static final String GROUP = "229.69.69.69" ;
8      public static final byte TTL = 1 ; // pas de saut!
9      public static final int DATA_MAX_SIZE = 512 ;
10
11     public static void main (String[] tbArgs) {
12         byte[] recBuffer = new byte[DATA_MAX_SIZE] ;
13         try{
14             MulticastSocket veille = new MulticastSocket(PORT);
15             InetAddress adrGroupe = InetAddress.getByName(GROUP) ;
16             while (veille != null) {
17                 ByteArrayOutputStream boz = new ByteArrayOutputStream();
18                 ObjectOutputStream oz = new ObjectOutputStream (boz) ;
19                 oz.writeObject(new Timestamp(System.currentTimeMillis())) ;
20                 // consommation excessive de données (deux byte buffers!)
21                 DatagramPacket sendPack =
22                     new DatagramPacket(boz.toByteArray(), boz.size(),
23                                         adrGroupe, PORT) ;
24                 veille.send(sendPack, TTL) ;
25
26                 try{
27                     Thread.sleep(1000) ;
28                 } catch (InterruptedException exc) {}
29             }
30         } catch (Exception exc ) {
31             System.err.println(exc) ;
32         }
33     } //End main
34 }
35

```



Exemple de client Multicast

Reçoit 10 objets du serveur

```
1  import java.io.* ;
2  import java.net.* ;
3  import java.sql.* ;
4
5  public class MultigClient {
6      public static final int DATA_MAX_SIZE = 512 ;
7
8      public static void main (String[] tbArgs) {
9          MulticastSocket socket = null;
10         InetAddress adrGroupe = null ;
11         byte[] recBuffer = new byte[DATA_MAX_SIZE] ;
12         try{
13             socket = new MulticastSocket (MultigServer.PORT) ;
14             adrGroupe = InetAddress .getByName (MultigServer.GROUP) ;
15             if (socket != null) {
16                 socket .joinGroup (adrGroupe) ;
17                 // danger! on reutilise le meme buffer?
18                 DatagramPacket recvPack =
19                     new DatagramPacket (recBuffer, recBuffer.length) ;
20                 for( int ix = 0 ; ix < 10 ; ix++) {
21                     socket.receive(recvPack) ;
22                     ObjectInputStream inz =
23                         new ObjectInputStream
24                             (new ByteArrayInputStream(
25                                 recvPack.getData())) ;
26                     Object obj = inz.readObject() ;
27                     System.out.println(obj) ;
28                     inz.close() ;
29                 }// for
30             }
31         } catch (Exception exc ) {
32             System.err.println(exc) ;
33         }
34         finally {
35             if ( socket != null) {
36                 try{ socket .leaveGroup (adrGroupe) ;
37                 } catch (IOException exc) {}
38             }
39         }
40     }
41 }
42
```

Approfondissements

** Voir dans le package les classes permettant de manipuler et d'utiliser des URL et en particulier URLConnection et ses dérivés (URLConnection, JarURLConnection,..). Dans le cas d'échanges http voir également la pratique des ContentHandler.*

** Voir les classes liées à la sécurité avancée sur réseau (NetPermission, Authenticator,..), l'utilisation de javax.net.ssl et de https.*

** Voir également les mécanismes de personnalisation des sockets (SocketImplFactory) en particulier avec RMI.*





Points essentiels :

Une catégorie particulière de classe d'entrée/sortie permet de lire ou d'écrire des instance d'objets sur un flot :

- ObjectInputStream et ObjectOutputStream
- particularités du comportement des Stream d'objets
- personnalisations des E/S d'objets.



Introduction

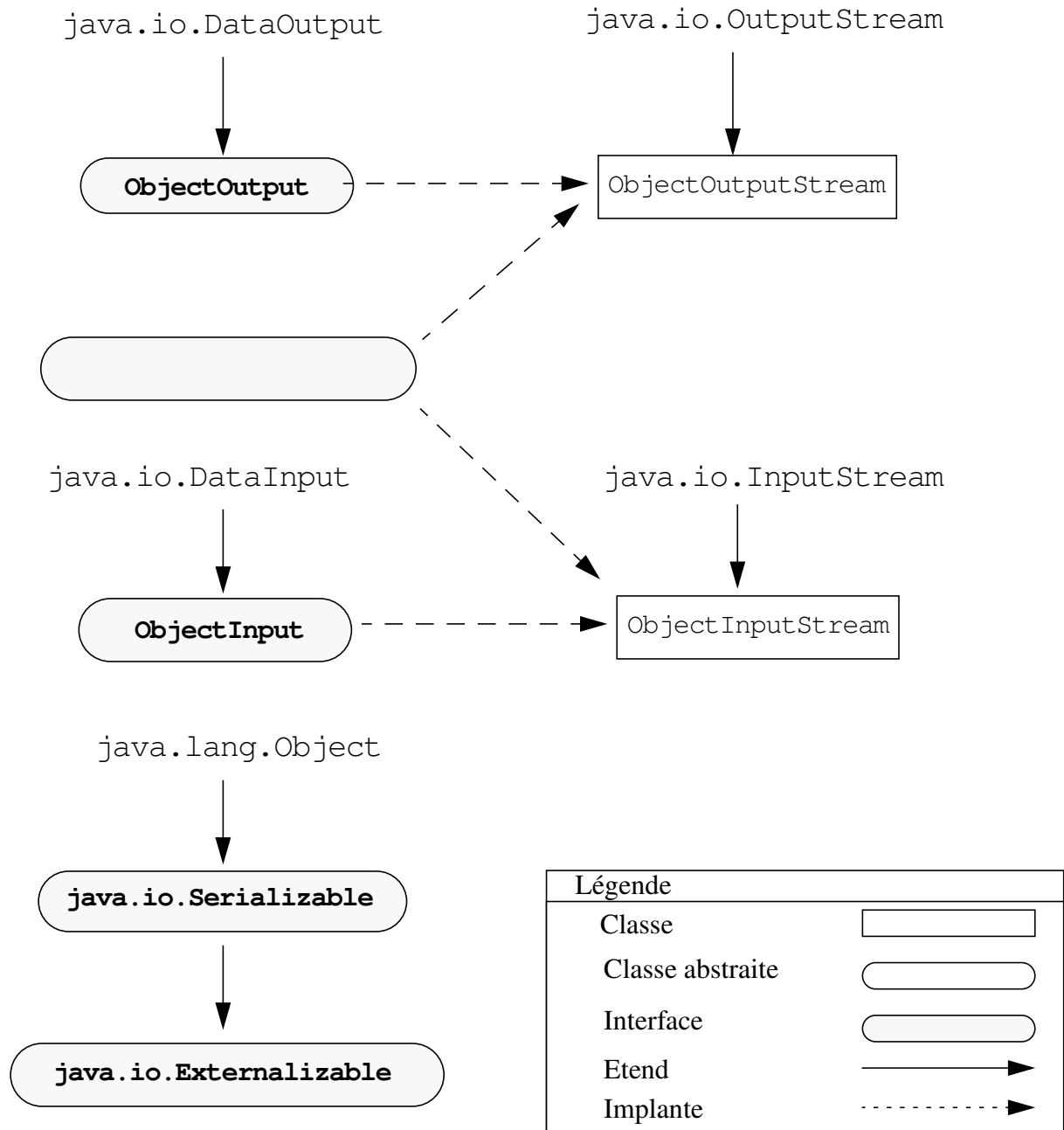
La majorité des applications requiert un outil de sauvegarde des données. La plupart de ces applications utilisent une base de données pour stocker ou conserver des données. Cependant, les bases de données ne servent pas uniquement à stocker des objets, particulièrement des objets Java. Pour les applications, le seul impératif est de conserver l'état d'un objet Java afin que cet objet puisse être facilement stocké et récupéré dans son état initial .

Dans cette optique, l' API de linéarisation d'objets Java fournit un moyen simple et transparent permettant de conserver les objets Java. Elle n'est pas utilisée uniquement pour sauvegarder ou restaurer des données mais sert aussi à échanger des objets sur des flots (par ex. échanges d'objets entre JVM distantes)

Architecture de sérialisation

Package `java.io`

L'API de sérialisation est basée sur deux interfaces abstraites à base de flots, `java.io.ObjectOutput` et `java.io.ObjectInput`, conçues pour introduire ou extraire des objets d'un flot d'E/S.





Architecture de sérialisation

Interface ObjectOutputStream

L'interface `ObjectOutputStream` étend `DataOutputStream` afin d'écrire des primitives. La principale méthode de cette interface est `writeObject()` qui permet d'écrire un objet. Des exceptions peuvent être générées lors de l'accès à l'objet ou à ses champs, ou lors d'une tentative d'écriture dans le flot.

```
package java.io;

public interface ObjectOutputStream extends DataOutputStream {

    public void writeObject(Object obj)
        throws IOException;

    public void write(byte b[]) throws IOException;

    public void write(byte b[], int off, int len)
        throws IOException;

    public void flush() throws IOException;

    public void close() throws IOException;
}
```


Architecture de sérialisation

Interface ObjectInput

La méthode `readObject` permet de lire le flot et de retourner un objet. Des exceptions sont générées lors d'une tentative de lecture du flot, ou s'il s'avère impossible de trouver le nom de classe pour l'objet sérialisé.

```
package java.io;
public interface ObjectInput extends DataInput {

    public Object readObject()
        throws ClassNotFoundException, IOException;

    public int read() throws IOException;

    public int read(byte b[]) throws IOException;

    public int read(byte b[], int off, int len)
        throws IOException;

    public long skip(long n) throws IOException;

    public int available() throws IOException;

    public void close() throws IOException;
}
```



Architecture de sérialisation

Interface Serializable

L'interface **Serializable** sert à identifier les classes pouvant être sérialisées :

```
package java.io;

public interface Serializable {};
```

Toute classe peut être sérialisée dans la mesure où elle satisfait aux critères suivants :

- La classe (ou une classe de la hiérarchie de cette classe) doit implémenter `java.io.Serializable`. Parmi les classes standard non-susceptibles d'être linéarisées citons `java.io.FileInputStream`, `java.io.FileOutputStream` et `java.lang.Threads`.
- Les champs à ne pas sérialiser doivent être repérés à l'aide du mot clé **transient**. Si le mot clé `transient` n'est pas affecté à ces champs, une tentative d'appel de la méthode `writeObject()` générera une exception `NotSerializableException`.

Eléments sérialisables

Tous les champs (données) d'un objet `Serializable` sont écrits dans le flot . Ils incluent les types de primitives, les tableaux et les références à d'autres objets. A nouveau, seules les données (et le nom de classe) des objets référencés sont stockées.

Les champs statiques ne sont pas sérialisés.

Il est à noter que le mécanisme d'accès aux champs (`private`, `protected` et `public`) n'a aucun effet sur le champ en cours de sérialisation. .

Ecriture et lecture d'un flot d'objets

Ecriture

Soit la classe :

```
class Point implements java.io.Serializable {
    int x;
    int y;
    Point( int x, int y){
        this.x = x;
        this.y = y;
    }
}
```

L'écriture et la lecture d'un objet dans un flot est un processus simple. Examinons le fragment de code suivant qui transmet une instance d'un objet à un fichier :

```
Point myPoint = new Point(1,2);
FileOutputStream fos = new FileOutputStream(myfile);
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(myPoint);
oos.close();
```



Écriture et lecture d'un flot d'objets

Lecture

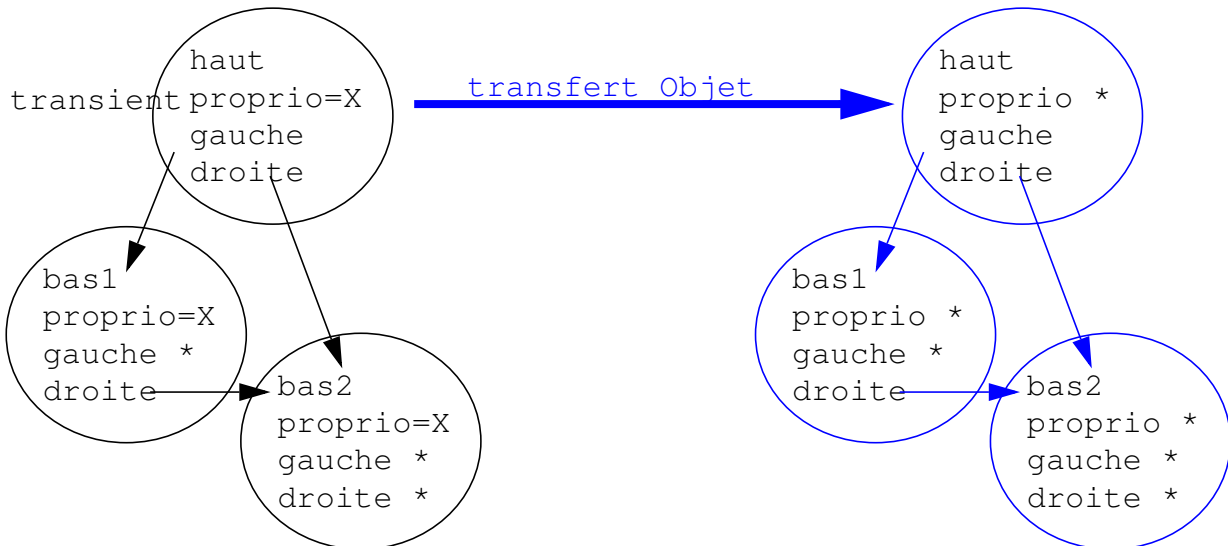
La lecture de l'objet est aussi simple que l'écriture, à une exception près — la méthode `readObject()` retourne un résultat de type `Object`, on doit donc transtyper (cast) le résultat, avant que l'exécution des méthodes sur cette classe soit possible :

```
Point serialPoint;  
FileInputStream fis = new FileInputStream(myFile);  
ObjectInputStream ois = new ObjectInputStream(fis);  
serialPoint = (Point)ois.readObject();
```

Effets de la linéarisation

Lorsqu'on linéarise des objets avec des ObjectStreams on doit bien maîtriser les effets suivants :

- Les instances référencées par l'instance en cours de linéarisation sont à leur tour linéarisées. **Attention:** les graphes de références sont conservés (y compris s'il y a des cycles!).



- Un mécanisme particulier (qui assure la propriété ci-dessus) fait que certaines méthodes ne sont appelées qu'**une fois** sur une instance donnée. Pour que le Stream "oublie" les instances déjà transférées utiliser la méthode `reset()`.
- La création d'un `ObjectInputStream` est un **appel bloquant**. On ne sortira de cet appel que lorsque le Stream aura reçu un premier transfert qui lui permet de s'assurer que le protocole avec l'`ObjectOutputStream` correspondant est correct.
- L'utilisation la plus simple du mécanisme de linéarisation suppose que les JVM qui écrivent et lisent les instances aient une connaissance *a priori* de la définition des classes. Il peut se produire des cas où les versions de définition de la classe ne sont pas tout à fait les mêmes entre les JVM : la spécification du langage Java (JLS) définit précisément les cas où ces versions sont considérés comme compatibles. Voir l'utilitaire `serialver` pour connaître l'identifiant de serialisation d'une classe.



Personnalisation de la linéarisation

personnalisation de la lecture/écriture d'objet

On peut personnaliser la linéarisation d'une classe en définissant deux méthodes privées `writeObject` and `readObject`.

La méthode `writeObject` permet de contrôler quelles informations sont sauvegardées. Typiquement on l'utilise pour envoyer des informations complémentaires qui permettront de reconstituer correctement l'objet à l'arrivée : supposons par exemple qu'on ait des deux cotés un mécanisme de dictionnaire (ou de base de données) qui permette de rechercher un objet en connaissant une clef; au lieu de transférer cet objet (qui est référencé par l'objet courant) on peut transférer sa clef et le reconstituer à l'arrivée en consultant le dictionnaire.

```
private void writeObject(ObjectOutputStream s)
    throws IOException {
    s.defaultWriteObject();
    // code spécifique : utiliser evt. les méthodes de
    // DataOutputStream pour les types scalaires et les chaînes
}
```

La méthode `readObject` doit être soigneusement conçue pour lire exactement les données dans le même ordre.

```
private void readObject(ObjectInputStream s)
    throws IOException {
    s.defaultReadObject();
    //lecture des données personnalisées
    ...
    // code de mise à jour de l'instance courante
}
```

Les méthodes `writeObject` et `readObject` ne sont responsables que de la linéarisation de la classe courante. Toute linéarisation de super-classe est traitée automatiquement. Si on a besoin de coordination explicite avec la super-classe il vaut mieux passer par le mécanisme de l'interface `Externalizable`.

Personnalisation de la linéarisation

Externalisable

Les classes implantant l'interface `Externalizable`, prennent la responsabilité du stockage et de la récupération de l'état de l'objet lui-même.

```
package java.io;

public interface Externalizable extends Serializable {

    public void writeExternal (ObjectOutput out)
        throws IOException;

    public void readExternal (ObjectInput in)
        throws IOException, ClassNotFoundException;
}
```

Les objets externalisables doivent :

- Implémenter l'interface `java.io.Externalizable`.
- Implanter une méthode `writeExternal` pour enregistrer l'état de l'objet. La méthode doit explicitement correspondre au supertype pour conserver son état.
- Implanter une méthode `readExternal` pour lire les données du flot et restaurer l'état de l'objet. La méthode doit explicitement correspondre au supertype pour conserver son état.
- Etre responsables du format défini en externe. Les méthodes `writeExternal` et `readExternal` sont uniquement responsables de ce format.

Des classes externalisables impliquent que la classe soit `Serializable`, mais vous devez fournir les méthodes de lecture et d'écriture d'objets. Aucune méthode n'est fournie par défaut.

ATTENTION: les mécanismes d'externalisation sont une menace pour la sécurité! Il est recommandé de les utiliser avec une extrême prudence!



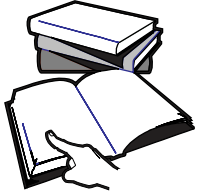


Points essentiels

Jusqu'à l'apparition de l' API RMI, les sockets constituaient l'unique fonction Java qui permettait d'établir une communication directe entre les machines. A l'instar des appels de procédures distants (RPC), l'architecture RMI utilise la connexion avec les sockets et les E/S pour le transfert des informations, si bien que les appels de méthodes sur des objets distants sont effectués de façon identique aux appels de méthodes sur des objets locaux.



Bibliographie



Certaines parties de ce module sont extraites de :

- “The Java Remote Method Invocation Specification” disponible sur

<http://chatsubo.javasoft.com/current/doc/rmi-spec/rmi-spec.ps>

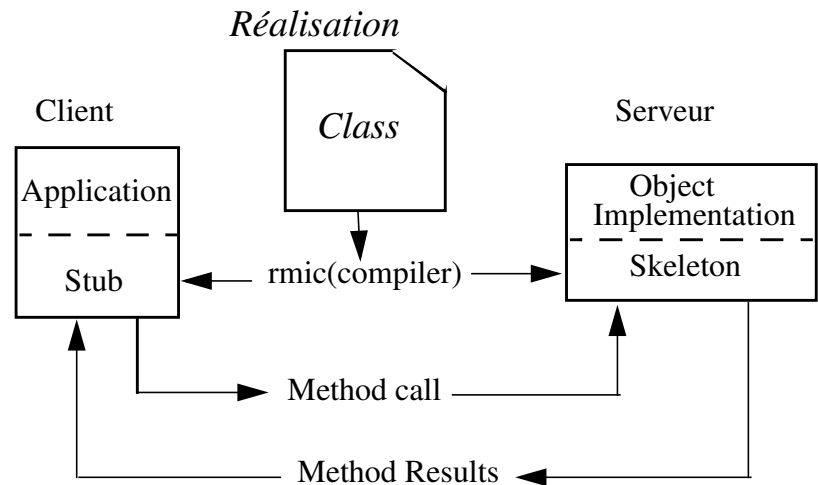
- “Java RMI Tutorial” disponible sur

<http://chatsubo.javasoft.com/current/doc/tutorial/rmi-getstart.ps>

- “Frequently Asked Questions, RMI and Object Serialization” disponible sur

<http://chatsubo.javasoft.com/current/faq.html>

Fonction de l'architecture RMI en Java

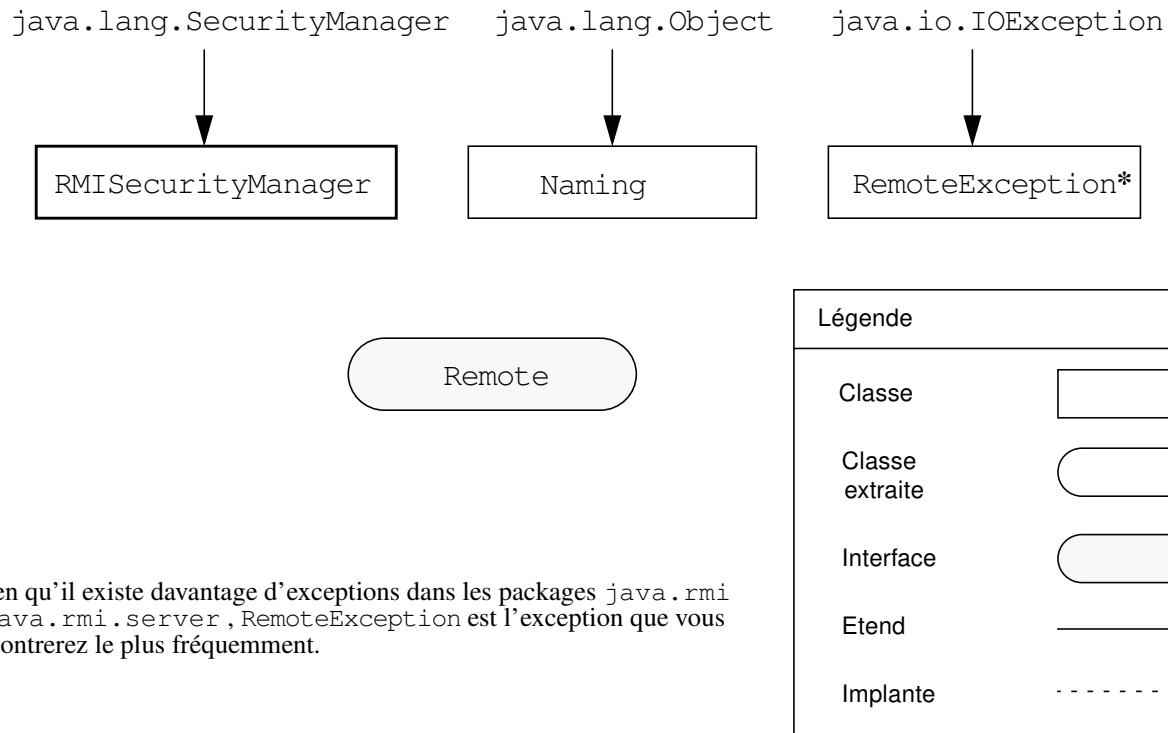


L' API RMI contient une série de classes et d'interfaces permettant au développeur d'appeler des objets distants, déjà existants dans une application s'exécutant sur une autre machine virtuelle Java (JVM). Cette JVM "distante" ou "serveur" peut être exécutée sur la même machine ou sur une machine entièrement différente du "client" RMI. L'architecture RMI en Java est un mécanisme utilisant uniquement le langage Java.



Packages et hiérarchies RMI

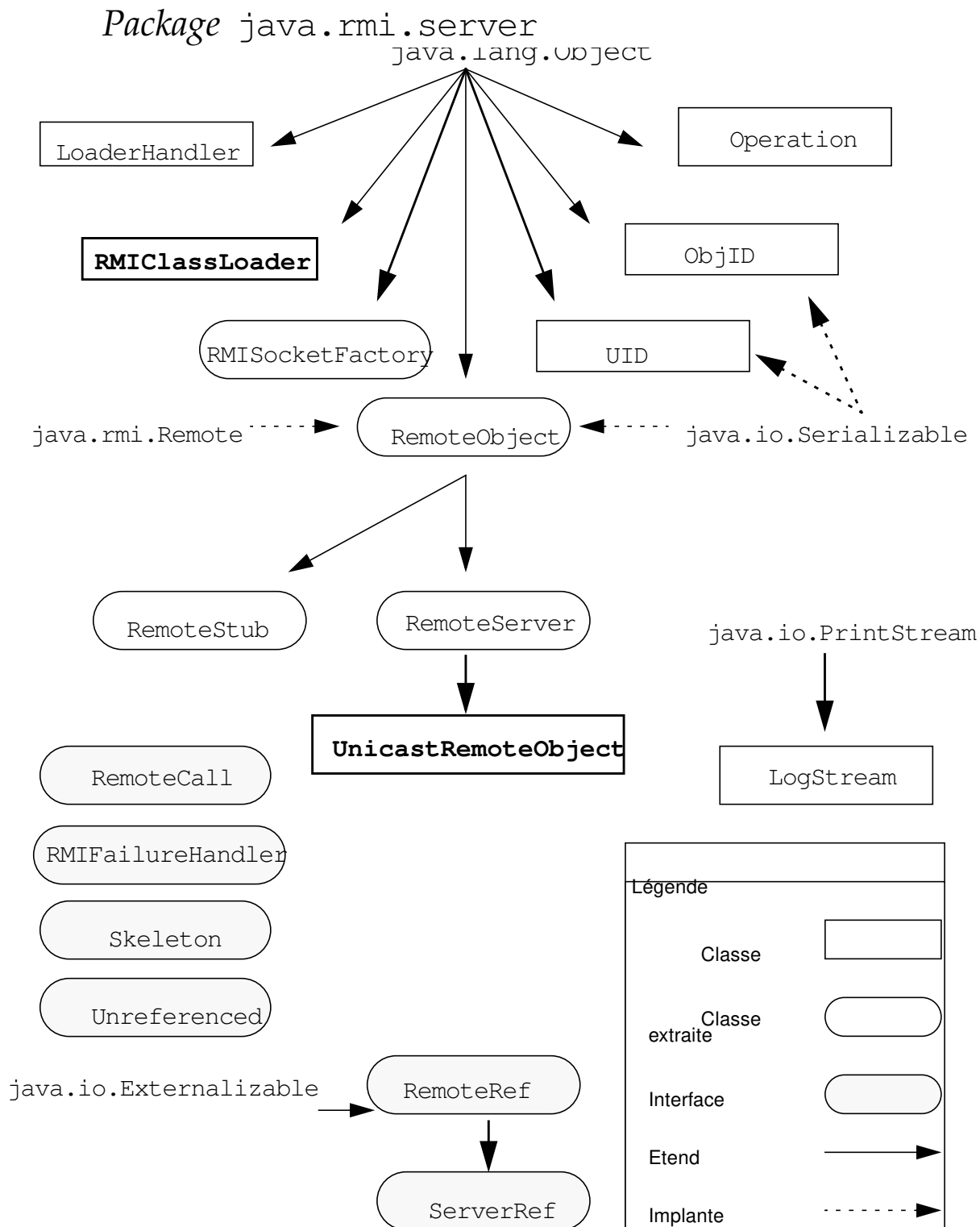
Package `java.rmi`



*Bien qu'il existe davantage d'exceptions dans les packages `java.rmi` et `java.rmi.server`, `RemoteException` est l'exception que vous rencontrerez le plus fréquemment.

- `Naming` – Cette classe “finale” est utilisée par les clients et serveurs RMI pour communiquer avec un “aiguilleur” appelé Registre des noms (`Registry`) et situé sur la machine serveur. L'application serveur utilise les méthodes `bind` et `rebind` pour enregistrer ses implantations d'objets auprès du Registre, alors que le programme client utilise la méthode `lookup` de cette classe pour obtenir une référence vers un objet distant.
- `Remote` – Cette interface doit être étendue par toutes les interfaces client qui seront utilisées pour accéder aux implantations d'objets distants.
- `RemoteException` – Cette exception doit être générée par toute méthode déclarée dans des interfaces et des classes de réalisation distantes. Tous les codes client doivent donc naturellement être écrits pour traiter cette exception.
- `RMISecurityManager` – Cette classe permet aux applications locales et distantes d'accéder aux classes et aux interfaces RMI.

Packages et hiérarchies RMI



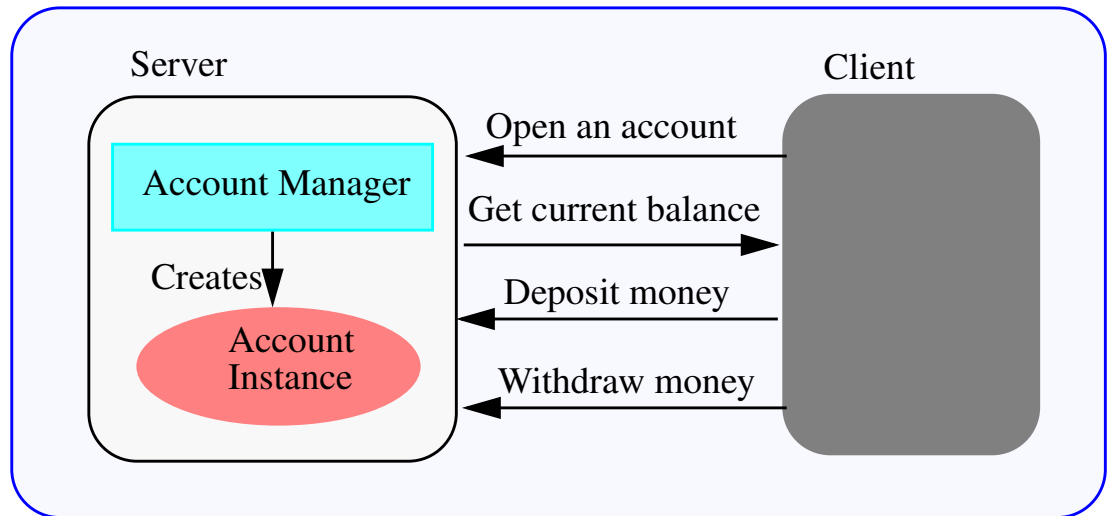


Packages et hiérarchies RMI

Package java.rmi.server (suite)

- `RMIClassLoader` – `ClassLoader` sert à charger les *stubs* (talons) et les *skeletons* d'objets distants, ainsi que les classes des arguments et les valeurs retournées par les appels de méthodes à distance. Lorsque `RMIClassloader` tente de charger des classes à partir du réseau, une exception est générée si aucun gestionnaire de sécurité n'est installé.
- `UnicastRemoteObject` – Classe parent de chaque classe distante en RMI

Création d'une application RMI



Exemple bancaire

Pour illustrer l'utilisation de l'invocation RMI, nous allons étudier l'exemple simple d'une banque dans laquelle on va ouvrir un compte. Les comptes sont contrôlés par un employé de banque : le gestionnaire de comptes.

Après avoir ouvert un compte, on peut y déposer ou en retirer de l'argent et vérifier le solde.



Création d'une application RMI

Interfaces bancaires

Si on tente de modéliser ce problème en utilisant l'invocation RMI, on peut créer deux interfaces Java du type suivant :

- Account.java

```
package rmi.bank;
interface Account extends Remote {
    public float getBalance ();
    public void withdraw (float money);
    public void deposit (float money);
}
```

- AccountManager.java

```
package rmi.bank;
interface AccountManager extends Remote {
    public Account open (String name,
        float startingBalance);
}
```


Création d'une application RMI

Interfaces bancaires (suite)

Il est à noter que ces interfaces sont conçues de sorte que l'on utilise l'interface `AccountManager` pour générer une instance d'un objet `Account`. `AccountManager` est chargé de retourner l'instance *courante* d'un objet `Account` si le compte existe déjà.

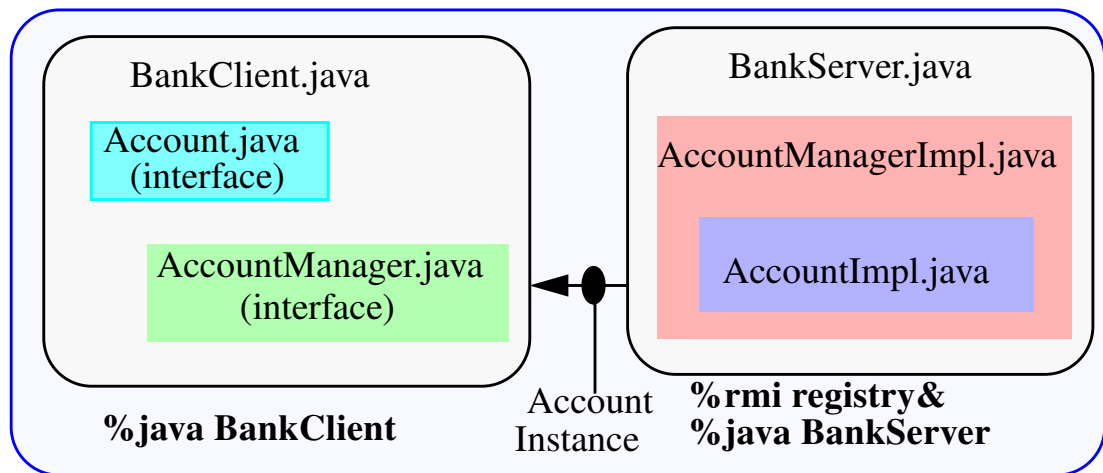
Cette approche de la création d'objet est un type de conception défini dans la méthodologie de programmation par une méthode appelée "Générateur (*Factory*)."¹

Ce "Générateur" permet à un objet de contrôler la création d'autres objets et, dans le cas considéré, il s'agit de la solution idéale car le gestionnaire de comptes (`AccountManager`) doit contrôler la création de nouveaux comptes. Si l'on se rendait dans la banque et on tentait d'ouvrir un autre compte, on répondrait "Vous possédez déjà un compte dans notre établissement. Utilisez-le."

1. *Design Patterns - Elements of Reusable Object-Oriented Software* par Gamma, Helm, Johnson et Vlissides (Éditions Addison-Wesley, 1995)



Création d'une application RMI



Procédure

Le processus permettant de créer une application accessible à distance dans RMI est le suivant :

1. Définir les objets distants à utiliser sous forme d'interfaces Java.
2. Créer des classes de réalisation pour les interfaces.
3. Compiler l'interface et les classes de réalisation.
4. Créer des classes *stubs* et *skeletons* à l'aide de la commande `rmic` sur les classes de réalisation.
5. Créer une application serveur permettant de gérer et de compiler les réalisations.
6. Créer et compiler un client permettant d'accéder aux objets distants.
7. Lancer `rmiregistry` et l'application serveur.
8. Tester le client.

Création d'une application RMI

Interface Account

L'exemple suivant présente l'interface Account complète :

```
// The Account interface
// Methods for getting the account balance, depositing,
// and
// withdrawing money
package rmi.bank;
import java.rmi.*;
public interface Account extends Remote {
    // Get the account balance
    public float getBalance () throws RemoteException;
    // Deposit money to the account -
    // throw an exception if the value
    // is 0 or a negative number
    public void deposit (float balance)
        throws BadMoneyException, RemoteException;
    // Withdraw money from the account -
    //but throw an exception if the
    // amount of the withdrawal will exceed the account
    balance
    public void withdraw (float balance)
        throws BadMoneyException, RemoteException;
}
```

L'interface Account doit étendre `java.rmi.Remote` et être déclarée `public` afin d'être rendue accessible à distance (aux clients utilisant d'autres JVM).

Il est à noter que toutes les méthodes génèrent une exception `java.rmi.RemoteException`. Cette exception apparaît lorsque survient un problème d'accès à la méthode d'un objet distant. Les méthodes de dépôt et de retrait génèrent également une exception `BadMoneyException` indiquant qu'une valeur négative a été transmise pour un dépôt, ou qu'une tentative de retrait supérieur au solde du compte a été effectuée.



Création d'une application RMI

Interface AccountManager

L'interface AccountManager complète inclut :

```
// The Account Manager interface

// Method for creating a new Account with the user's
// name and initial account balance

package rmi.bank;

import java.rmi.*;

public interface AccountManager extends Remote {

    public Account open (String name, float
        initialBalance)
        throws BadMoneyException, RemoteException;

}
```

Création d'une application RMI

réalisation de l'interface Account — AccountImpl

L'interface Account est réalisée par une classe devant à son tour implanter toutes les méthodes définies dans Account et étendre UnicastRemoteObject. Par convention, pour les classes réalisant des contrats d'interface RMI, le suffixe "Impl" sera ajouté au nom du fichier d'interface :

```
// AccountImpl - Implementation of the Account interface
//
// This class is an instance of an Account and implements
// the balance, deposit and withdraw methods specified by
// the Account interface.

package rmi.bank;

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class AccountImpl
    extends UnicastRemoteObject
    implements Account {

    // The current balance of the account
    private float balance = 0;

    // Create a new account implementation with a new
    account balance
    public AccountImpl (float newBalance)
        throws RemoteException {
        balance = newBalance;
    }
}
```



Création d'une application RMI

réalisation de l'interface Account — AccountImpl (suite)

```
// Methods implemented from Account

// Return the current account balance
public float getBalance () throws RemoteException {
    return balance;
}

// Deposit money into the account,
// as long as it is a positive number
public void deposit (float money)
    throws BadMoneyException, RemoteException {
    // Is the deposit amount a negative number?
    if (money < 0) {
        throw new BadMoneyException
            ("Attempt to deposit negative money!");
    } else {
        balance += money;
    }
}

// Withdraw money from the account, up to the
// value of the current account balance
public void withdraw (float money)
    throws BadMoneyException, RemoteException {
    // Is the deposit amount a negative number?
    if (money < 0) {
        throw new BadMoneyException
            ("Attempt to deposit negative money!");
    } else {
        // Is there sufficient money in the account?
        if ((balance - money) < 0) {
            throw new BadMoneyException
                ("Attempt to overdraw your account!");
        } else {
            balance -= money;
        }
    }
}
}
```

Création d'une application RMI

AccountManagerImpl

La classe `AccountManagerImpl` est chargée de créer et d'enregistrer de nouveaux comptes (sous forme d'objets `AccountImpl`). Cette classe utilise un vecteur (`Vector`) permettant d'enregistrer les objets `Account` dans une classe de conteneurs appelée `AccountInfo`, associée au nom du compte `String`. Cette classe utilitaire facilite la recherche d'un objet `Account` existant.

```
// AccountManagerImpl - Implementation of the
AccountManager
// interface
//
// This version of the AccountManager class stores all
// instances of the Account(s) it creates in a Vector
// object - if an account requested exists, it will
// return that account.

package rmi.bank;

import java.util.Vector;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class AccountManagerImpl
    extends UnicastRemoteObject
    implements AccountManager {

    // Local storage of account names
    private static Vector accounts = new Vector ();

    // This empty constructor is required to create an
    // instance of this class in the server
    public AccountManagerImpl () throws RemoteException
    {
    }
}
```



Création d'une application RMI

AccountManagerImpl (*suite*)

```
// Implement method from AccountManager interface
// Create an instance of an Account - if the account
// name already exists, return that account instead
// of creating a new one
public Account open (String name, float
initialBalance)
    throws BadMoneyException, RemoteException {
    AccountInfo a;
    // Check if this name is in the list already
    for (int i = 0; i < accounts.size(); i++) {
        a = (AccountInfo)accounts.elementAt(i);
        if (a.name.equals (name)) {
            return (a.account);
        }
    }
    // Check the initial account value...
    if (initialBalance < 0) {
        throw new BadMoneyException ("Negative initial
balance!");
    }
    // Store the new account
    a = new AccountInfo();
    // Try to create a new account with the starting
balance
    try {
        a.account = new AccountImpl (initialBalance);
    } catch (RemoteException e) {
        System.err.println ("Error opening account: "
+ e.getMessage());
        throw (e);
    }
    a.name = name;
    accounts.addElement (a);
    // Return and instance of an AccountImpl object
    return (a.account);
}
}
```


Création d'une application RMI

AccountManagerImpl

Classe de conteneurs

La classe de conteneurs AccountInfo est utilisée par la classe AccountManagerImpl.

```
// A container class for instance of Accounts
// that is stored in the Vector object

class AccountInfo {
    String name;
    AccountImpl account = null;
}
```



Création d'une application RMI

Compilation du code

Le chemin d'accès aux classes est important pour la réussite de l'exécution du code RMI. Il est recommandé d'envisager l'utilisation de l'option `-d` du compilateur pour localiser les fichiers de classes créés :

```
% javac -d classDirectory *.java
```

Dans l'exemple de code ci-dessus, le répertoire de package `rmi/bank` est créé dans le répertoire courant.

Création d'une application RMI

Utilisation de la commande `rmic`

Après compilation des classes de réalisation, vous devez créer les codes *stub* et *skeleton* permettant d'accéder aux classes de réalisation. Vous rappellerez les classes *stubs* utilisées par le code client pour communiquer avec le code *skeleton* du serveur.

La commande `rmic` créera des codes *stub* et *skeleton* à partir des définitions des interfaces et des classes de réalisation.

Cette étape doit être exécutée après compilation des classes de réalisation et avant exécution de l'application serveur. La syntaxe de la commande est la suivante :

```
rmic [options] package.interfaceImpl ...
```

Exemple :

```
% rmic -d classDirectory rmi.bank.AccountManagerImpl \  
rmi.bank.AccountImpl
```

L'exemple suivant créera quatre classes supplémentaires dans le répertoire `rmi/bank` (package) :

```
AccountImpl_Skel.class  
AccountImpl_Stub.class  
AccountManagerImpl_Skel.class  
AccountManagerImpl_Stub.class
```



Création d'une application RMI

Application BankServer

BankServer gère l'objet AccountManagerImpl. Son unique fonction consiste à fournir une instance AccountManager à tout client demandeur.

```
// BankServer - This class is run on the RMI server
// and is responsible for registering the AccountManager
// implementation.
package rmi.bank;
import java.rmi.*;
public class BankServer {
public static void main(String args[])
    {
        // Create and install the security manager
        System.setSecurityManager(new
RMISecurityManager());
        try {
            // Create the object instance for registration
            System.out.println
                ("BankServer.main: creating an
AccountManagerImpl");
            AccountManagerImpl acm = new AccountManagerImpl
                ();
            // Bind the object instance to the registry
            System.out.println
                ("BankServer.main: bind it to a name:
bankManager");
            Naming.rebind("bankManager", acm);
            System.out.println("bankManager Server
ready.");
        } catch (Exception e) {
            System.out.println
                ("BankServer.main: an exception occurred: " +
                e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Création d'une application RMI

Application BankServer (suite)

Le serveur "publie" l'instance de l'objet `AccountManagerImpl` en associant cet objet à un nom stocké dans une application "d'aiguillage" `rmiregistry`.

L'affectation s'effectue à la ligne 23 du programme précédent

```
Naming.rebind("bankManager", acm);
```

par le biais de la méthode `rebind` de la classe `java.rmi.Naming`. Cette méthode associe ou "affecte" le nom `bankManager` à l'objet `acm`, en supprimant tout objet précédemment affecté à ce nom dans le Registre.

La classe `Naming` fournit deux méthodes permettant au développeur d'enregistrer une réalisation, `bind` et `rebind`, la seule différence résidant dans le fait que la méthode `bind` générera une exception `java.rmi.AlreadyBoundException` si un autre objet a déjà été enregistré sur ce serveur, à l'aide du nom transmis à la méthode en tant que premier argument.

Les arguments d'affectation et de réaffectation se présentent sous forme de chaîne de type URL contenant le nom d'instance de réalisation d'objet. La chaîne URL doit respecter le format

```
rmi://host:port/name
```

où `rmi` désigne le protocole, `host` est le nom du serveur RMI (qui devrait être compatible DNS ou NIS+), `port` est le numéro de port que le serveur doit écouter pour les requêtes, et `name` est le nom exact que les clients doivent utiliser dans les requêtes `Naming.lookup` pour cet objet.

Les valeurs par défaut sont `rmi` pour le protocole, l'hôte local pour `host` 1099 pour `port`.



Création d'une application RMI

Application `rmiregistry`

`rmiregistry` est une application fournissant un simple service d'aiguillage sur un nom. L'application `BankServer` fournit à l'application `rmiregistry` la référence d'objet et un `nom` string par le biais de l'appel de méthode `rebind`.

L'application `rmiregistry` doit être exécutée avant que l'application `BankServer` ne tente l'affectation :

```
% rmiregistry &  
% java rmi.bank.BankServer &
```

Les propriétés peuvent être définies pour la machine JVM du serveur RMI dans la ligne de commande :

- `java.rmi.server.codebase` – Cet URL indique l'emplacement où les clients peuvent télécharger les classes.
- `java.rmi.server.logCalls` – Si la valeur retournée est "vraie", le serveur consigne les appels dans `stderr`. La valeur par défaut est "faux".

```
% java -Djava.rmi.server.logCalls=true rmi.bank.BankServer &
```

Après exportation de la réalisation par le Registre, le client peut expédier une chaîne URL pour demander à ce que l'application `rmiregistry` fournisse une référence de l'objet distant. La recherche s'effectue par le biais d'un appel client de `Naming.lookup`, en transmettant une chaîne URL sous forme d'argument :

```
rmi://host:port/name
```

La page suivante présente la recherche utilisée par l'application client.

Création d'une application RMI

Application BankClient

L'application `BankClient` tente de localiser un objet `AccountManager` en effectuant une recherche à l'aide d'un aiguilleur (`Registry`). Cet aiguilleur se situe dans `host:port` dans la chaîne URL transmise à la méthode `Naming.lookup()`. L'objet retourné est "vu" (converti par `cast`) comme un gestionnaire de compte (`AccountManager`) et peut servir à ouvrir un compte avec un nom et à lancer le calcul du solde.

```
// BankClient - the test program for the Bank RMI example
//
// This class simply attempts to locate the "bankManager"
// RMI object reference, then binds to it and opens an
// instance to an AccountManager implementation at the
// <server> location.
//
// Then it requests an Account with <name> and
// optionally,
// an initial balance (for a new account).
//
// The class then tests the account by depositing and
// withdrawing money and looking at the account balance.
package rmi.bank;
import java.rmi.*;
public class BankClient {
public static void main(String args[])
{
    // Check the argument count
    if (args.length < 2) {
        System.err.println ("Usage:");
        System.err.println
            ("java BankClient <server> <account name>
[initial balance]");
        System.exit (1);
    }
    // Create and install the security manager
    System.setSecurityManager(new
RMISecurityManager());
}
```



Création d'une application RMI

Application BankClient (suite)

```
try {
    // Get the bank instance
    System.out.println ("BankClient: lookup
bankManager");
    String url = new String
("rmi://" + args[0] + "/bankManager");
    AccountManager acm =
(AccountManager)Naming.lookup(url);
    // Set the account balance, if passed as an
argument
    float startBalance = 0.0f;
    if (args.length == 3) {
        Float F = Float.valueOf(args[2]);
        startBalance = F.floatValue();
    }
    // Get an account (either new or existing)
    Account account = acm.open (args[1],
startBalance);
    // Now do some stuff with the remote object
implementation
    System.out.println ("BankClient: current balance
is: " +
        account.getBalance ());
    System.out.println ("BankClient: withdraw
50.00");
    account.withdraw (50.00f);
    System.out.println ("BankClient: current balance
is: " +
        account.getBalance ());
    System.out.println ("BankClient: deposit
100.00");
    account.deposit (100.00f);
    System.out.println ("BankClient: current balance
is: " +
        account.getBalance ());
    System.out.println ("BankClient: deposit 25.00");
    account.deposit (25.00f);
    System.out.println ("BankClient: current balance
is: " +
        account.getBalance ());
```



```
        } catch (Exception e) {
            System.err.println("BankClient: an exception
occurred: " +
                e.getMessage());
            e.printStackTrace();
        }
        System.exit(1);
    }
}
```



Création d'une application RMI

Application BankClient (suite)

Après avoir ouvert un compte, l'application `BankClient` exécute les opérations simples de dépôt et de retrait. Cette classe pourrait (et devrait probablement) posséder une interface interactive qui permettrait au client de saisir le nom du compte, puis d'effectuer un retrait ou un dépôt.

Exécution de l'application BankClient

L'application `BankClient` peut être exécutée à partir de tout hôte autorisé à accéder au Registre et au package contenant les fichiers de classes de l'application client, des stubs et des interfaces.

Création d'une application RMI

Exécution de l'application BankClient

Syntaxe

```
java rmi.bank.BankClient hostname accountName initialBalance
```

Exemples

```
% java rmi.bank.BankClient mach1 fred 1000
BankClient: lookup bankManager
BankClient: current balance is: 1000.0
BankClient: withdraw 50.00
BankClient: current balance is: 950.0
BankClient: deposit 100.00
BankClient: current balance is: 1050.0
BankClient: deposit 25.00
BankClient: current balance is: 1075.0
```

Il est à noter que l'objet distant `AccountManagerImpl` situé sur le serveur, enregistre l'instance du compte créé avec le nom `fred`. Par conséquent, la réexécution de l'application `BankClient` avec le même nom de compte utilisera le même objet :

```
% java rmi.bank.BankClient mach1 fred
BankClient: lookup bankManager
BankClient: current balance is: 1075.0
BankClient: withdraw 50.00
BankClient: current balance is: 1025.0
BankClient: deposit 100.00
BankClient: current balance is: 1125.0
BankClient: deposit 25.00
BankClient: current balance is: 1150.0
```



Sécurité RMI

Chargement de classe

Pour utiliser `RMIClassLoader`, un gestionnaire de sécurité doit déjà exister afin de s'assurer que les classes chargées à partir du réseau satisfont aux critères standard de sécurité Java. Si aucun gestionnaire n'est en place, l'application ne peut pas charger les classes à partir d'hôtes distants.

Politique de sécurité

De fait le `RMISecurityManager` n'est strictement nécessaire que s'il y a chargement de code. La présence de ce gestionnaire de sécurité rend obligatoire l'utilisation d'un fichier `policy` :

```
java -Djava.security.policy=myrmi.policy rmi.bank.Client .....
```

Ce fichier `policy` contenant une entrée de type :

```
grant {  
permission java.net.SocketPermission "host:1024-", "connect" ;  
} ;
```

On peut également tenter de réduire le nombre de ports admis en spécifiant : le port du `rmiregistry`, un port que les objets serveurs ont choisi (voir constructeur spécial de `UnicastRemoteObject`).

Sécurité RMI

Chargement de classe

Côté client RMI

Si le programme client RMI est une applet, son gestionnaire de sécurité (`SecurityManager`) et son chargeur de classe (`ClassLoader`) sont mandatés par le browser côté client.

Cependant, si le programme client est une application, les seules classes qui seraient téléchargées à partir du serveur RMI seraient les définitions d'interfaces distantes, les classes stubs, les classes d'arguments étendues et les valeurs retournées par les appels de méthodes distants. Si une application client tente de charger des classes supplémentaires à partir du serveur, elle peut utiliser `RMIClassLoader.loadClass`, en fournissant comme paramètres les mêmes URL et identificateur que ceux transmis dans `Naming.lookup`.

Invocation RMI au travers d'un coupe-feu

La couche transport RMI tente normalement d'ouvrir des sockets directs des clients aux serveurs. Néanmoins, s'il est impossible d'établir une connexion directe au serveur par socket, la méthode `createSocket` de la classe `java.rmi.server.RMISocketFactory` retentera la requête sous forme de connexion par protocole de transfert hypertexte (HTTP) en expédiant l'appel RMI sous forme de requête HTTP POST.

Si la méthode `createServerSocket` détecte que la connexion récemment acceptée est une requête HTTP POST, les informations retournées seront réexpédiées dans le corps d'une réponse HTTP.

Aucune configuration spéciale n'est requise pour permettre au client d'effectuer des appels RMI au travers d'un coupe-feu.



Approfondissements

** RMI et politiques de sécurité; téléchargement des talons "stubs", callback des clients, serveur Activatable et rmid, RMI et IIOP.*



Points essentiels :

L'API JDBC contient une série d'interfaces conçues pour permettre au développeur d'applications qui travaillent sur des bases de données de le faire indépendamment du type de base utilisé

- Pilotes (drivers) JDBC.
- établissement de connexion et contexte d'exécution (Statement)
- requêtes à la base et récupération des résultats.

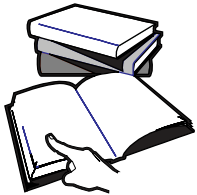


Introduction

La connectivité JDBC permet au développeur de se concentrer sur l'écriture de l'application en s'assurant que les interrogations de la base de données sont correctes et que les données sont manipulées conformément à leur conception.

La connectivité JDBC permet au développeur d'écrire une application en utilisant les noms d'interfaces et les méthodes décrites dans l'API, sans tenir compte de leur réalisation dans le pilote (driver JDBC). Le développeur utilise les interfaces décrites dans l'API comme s'il s'agissait de classes courantes. Le constructeur du pilote fournit une réalisation de classe pour chaque interface de l'API. Lorsqu'une méthode d'interface est utilisée, elle se réfère en fait à une instance d'objet d'une classe ayant réalisé cette interface.

Bibliographie



- Caractéristiques JDBC : <http://java.sun.com/products/jdbc>
- *The Practical SQL Handbook* par Emerson, Darnovsky et Bowman (Editions Addison-Wesley, 1989)

“Pilote” JDBC

Chaque pilote de base de données doit fournir une classe réalisant l’interface `java.sql.Driver`. Cette classe est alors utilisée par la classe générique `java.sql.DriverManager` lorsqu’un pilote est requis pour assurer la connexion à une base de données spécifique. Cette connexion s’opère à l’aide d’une URL (identificateur de ressources).

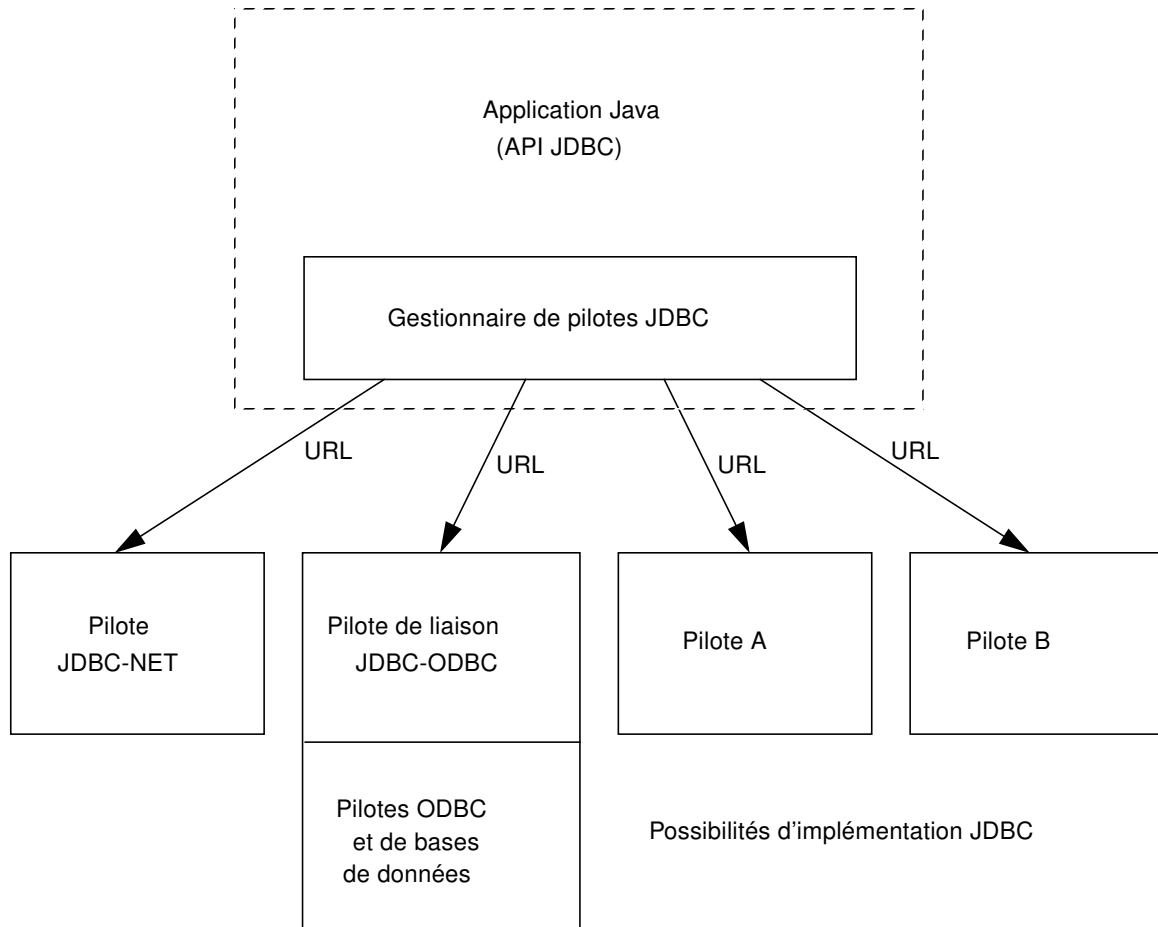
Exemple de driver (utilisé dans les exercices) : `com.imaginary.sql.mssql.MssqlDriver`¹, un pilote JDBC écrit pour une connexion à une base de données Mini-SQL². Le pilote Imaginary illustre la flexibilité du langage Java.

1. API mSQL-JDBC fournie avec l’aimable autorisation de George Reese.<http://www.imaginary.com/~borg>
2. Mini SQL fourni avec l’aimable autorisation de Hughes Technologies Pty Ltd, Australie.





Pilotes JDBC



Organigramme JDBC

L'enchaînement des appels

Du point de vue du programmeur JDBC les tâches s'enchainent de la manière suivante :

- Création d'une instance d'un driver JDBC.
- détermination de la base
- Ouverture d'une connexion à la base
- Allocation d'un contexte de requête (Statement)
- Soumission d'une requête
- Récupération des résultats

Package java.sql

Huit interfaces sont associées à l'API JDBC :

- Driver
- Connection
- Statement
- PreparedStatement
- CallableStatement
- ResultSet
- ResultSetMetaData
- DatabaseMetaData



Organigramme JDBC

Chacune de ces interfaces permet à un programmeur d'application d'établir des connexions à des bases de données spécifiques, d'exécuter des instructions SQL et de traiter les résultats.

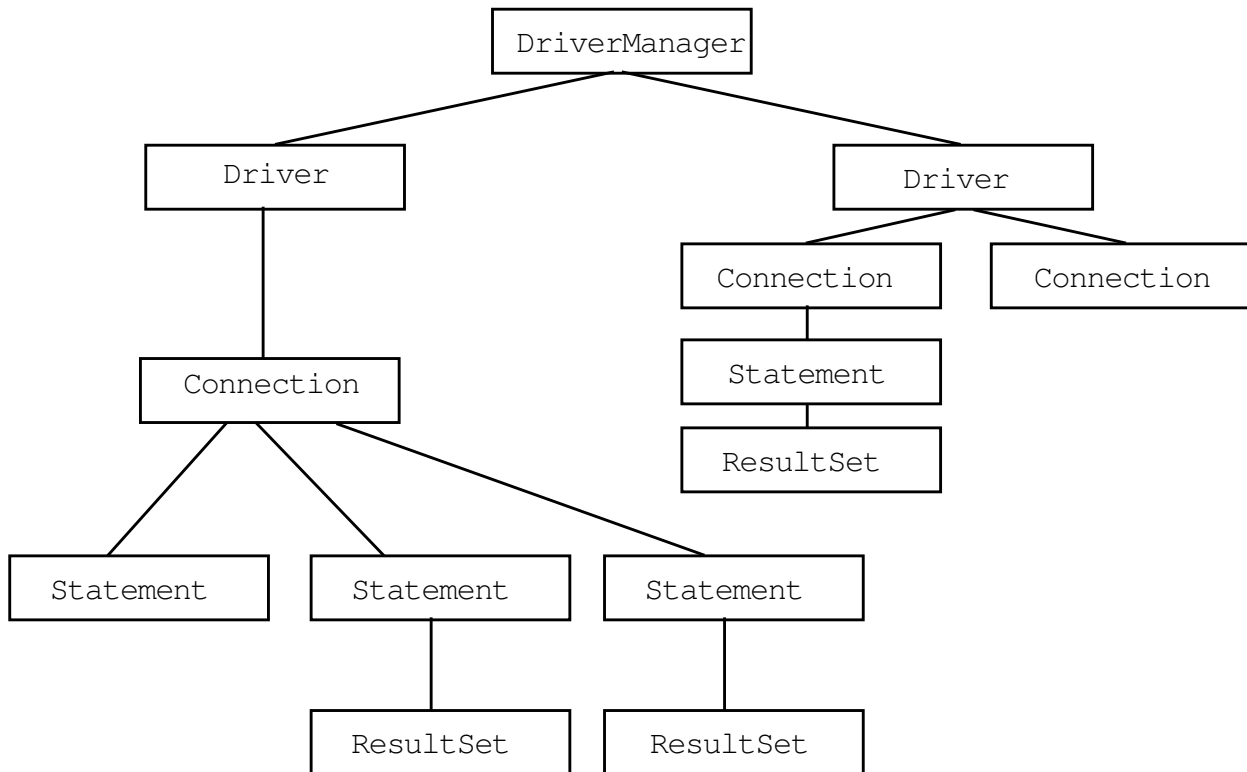


Figure 0-1 Organigramme JDBC

- Une chaîne d'URL est transmise à la méthode `getConnection()` du gestionnaire de pilotes (`DriverManager`) qui localise à son tour un pilote (`Driver`).
- Un pilote vous permet d'obtenir une connexion (`Connection`).
- Cette connexion vous permet de créer une requête (`Statement`).
- Lorsqu'une requête est exécutée avec une méthode `executeQuery()`, un résultat (`ResultSet`) peut être retourné.

Exemple JDBC

Cet exemple simple utilise la base de données Mini-SQL, ainsi que les éléments d'une application JDBC. Les opérations réalisées seront les suivantes : création d'une instance `Driver`, obtention d'un objet `Connection`, création d'un objet `Statement` et exécution d'une requête, puis traitement de l'objet retourné `ResultSet`.

```
1 import java.sql.*;
2 import com.imaginary.sql.msql.*;
3
4 public class JDBCExample {
5
6     public static void main (String args[]) {
7
8         if (args.length < 1) {
9             System.err.println ("Usage:");
10            System.err.println (" java JDBCExample <db server hostname>");
11            System.exit (1);
12        }
13        String serverName = args[0];
14        try {
15            // Create the instance of the Msql Driver
16            new MsqlDriver ();
17
18            // Create the "url"
19            String url = "jdbc:mssql://" + serverName +
20                ":1112/StockMarket";
21
22            // Use the DriverManager to get a Connection
23            Connection mSQLcon = DriverManager.getConnection (url);
24
25            // Use the Connection to create a Statement object
26            Statement stmt = mSQLcon.createStatement ();
27
28            // Execute a query using the Statement and return a ResultSet
29            ResultSet rs = stmt.executeQuery (
30                "SELECT ssn, cust_name FROM Customer" +
31                " order by cust_name");
```



Exemple JDBC

```
32         // Print the results, row by row
33         while (rs.next()) {
34             System.out.println ("");
35             System.out.println ("Customer: " + rs.getString (2));
36             System.out.println ("Id:          " + rs.getString (1));
37         }
38
39     } catch (SQLException e) {
40         e.printStackTrace();
41     }
42 }
43 }
```

Résultats:

```
% java JDBCExample serveur

Customer: Tom McGinn
Id:          999-11-2222

Customer: Jennifer Sullivan Volpe
Id:          999-22-3333

Customer: Georgianna DG Meagher
Id:          999-33-4444

Customer: Priscilla Malcolm
Id:          999-44-5555
```

Création de pilotes JDBC

Il existe deux méthodes pour créer une instance de pilote JDBC : explicitement ou à l'aide de la propriété `jdbc.drivers`.

Création explicite d'une instance de pilote JDBC

Pour communiquer avec un moteur de base de données particulier en JDBC, vous devez préalablement créer une instance pour le pilote JDBC. Ce pilote reste en arrière plan et traite toutes les requêtes pour ce type de base de données.

```
// Create an instance of Mysql's JDBC Driver  
new MySQLDriver();
```

Il n'est pas nécessaire d'associer ce pilote à une variable car le pilote est référencé par un objet statique.



Création de pilotes JDBC

Chargement des pilotes JDBC via `jdbc.drivers`

Il est tout à fait possible que plusieurs pilotes de bases de données soient chargés en mémoire. Il peut également arriver que plusieurs de ces pilotes, ODBC ou protocoles génériques de réseau JDBC, soient en mesure de se connecter à la même base de données. Dans ce cas, l'interface JDBC permet aux utilisateurs de définir une liste de pilotes dans un ordre spécifique. Cet ordre de sélection est défini par un paramètre de propriétés Java, `jdbc.drivers`. La propriété `jdbc.drivers` doit être définie sous forme de liste de noms de classes de pilotes, séparés par le symbole deux points (":") :

```
jdbc.drivers=com.imaginary.sql.mssql.MssqlDriver:com.acme.wonder.driver
```

Les propriétés sont définies par l'option `-D` de l'interpréteur `java` (ou l'option `-J` de l'application `appletviewer`). Exemple :

```
% java -Djdbc.drivers=com.imaginary.sql.mssql.MssqlDriver:\  
com.acme.wonder.driver
```

Lors d'une tentative de connexion à une base de données, l'API JDBC utilise le premier pilote trouvé susceptible d'établir la connexion à l'URL défini. L'API essaie tout d'abord chaque pilote défini dans cette propriété, dans l'ordre de gauche à droite. Elle essaie ensuite tous les pilotes déjà chargés en mémoire en respectant l'ordre de chargement. Si le pilote a été chargé par un code non sécurisé, il est alors ignoré sauf s'il a été chargé à partir de la même source que le code tentant d'établir la connexion.

Pilotes JDBC

Désignation d'une base de données

Après avoir créé l'instance du pilote JDBC, vous devez à présent indiquer la base de données à laquelle vous souhaitez vous connecter.

Dans JDBC, il vous suffit de spécifier un URL indiquant le type de base de données. La syntaxe des chaînes d'URL proposée pour une base de données JDBC est :

```
jdbc:sous_protocole:parametres
```

`sous_protocole` désigne un type spécifique de mécanisme de connectivité des bases de données, pouvant être supporté par un ou plusieurs pilotes. Le contenu et la syntaxe de `parametres` dépendent du sous-protocole.

```
// Construct the URL for JDBC access
String url = new String ("jdbc:mysql://" + servername
    + ":1112/StockMarket");
```

Cet URL vous permettra d'accéder à la base de données mSQL StockMarket . `servername` est une variable définissant le nom d'hôte du serveur de la base de données.



Connexion JDBC

Connexion à une base de données

Après avoir créé un URL définissant `msql` en tant que moteur de base de données, vous pouvez à présent établir une connexion à la base de données.

A cet effet, on doit obtenir un objet `java.sql.Connection` en appelant la méthode `java.sql.DriverManager.getConnection` du pilote JDBC.

```
// Establish a database connection through the msql
// DriverManager
Connection mSQLcon =
DriverManager.getConnection(url);
```

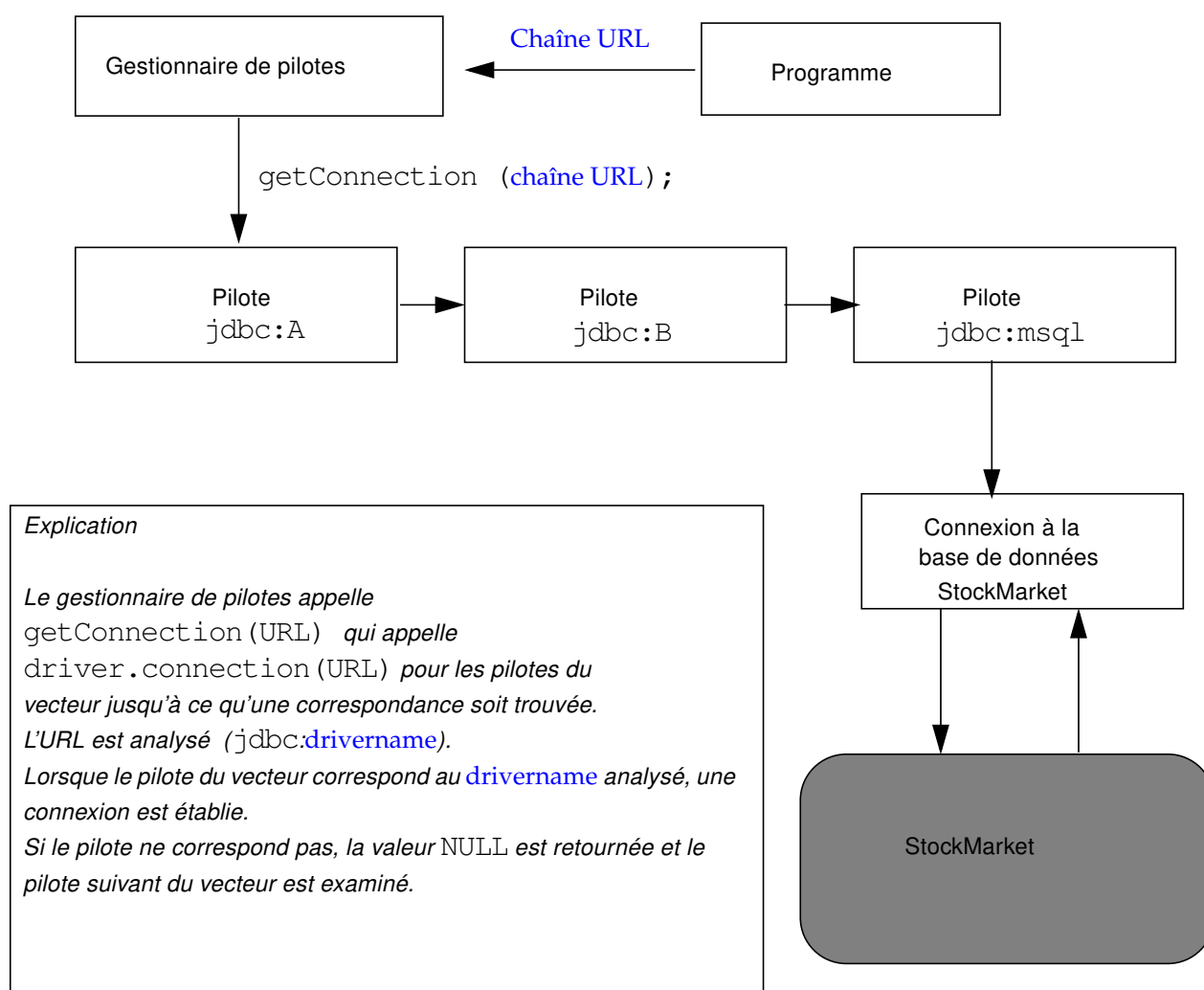
Le processus est le suivant :

- Le gestionnaire de pilotes (`DriverManager`) appelle la méthode `Driver.getConnection` pour chaque pilote enregistré, en transmettant la chaîne d'URL sous forme de paramètre.
- Si le pilote identifie le nom du sous-protocole, il retourne alors une instance d'objet `Connection` ou une valeur nulle le cas échéant.

Pilotes JDBC

Interrogation d'une base de données

La figure 2-4 décrit la méthode permettant à un gestionnaire de pilotes (DriverManager) de traduire une chaîne URL transmise dans la méthode `getConnection()`. Lorsque le pilote retourne une valeur nulle, le gestionnaire appelle le pilote enregistré suivant jusqu'à la fin de la liste ou jusqu'à ce qu'un objet `Connection` soit retourné.





Instructions JDBC

Soumission d'une requête

Pour soumettre une requête standard, créez tout d'abord un objet Statement à partir de la méthode `Connection.createStatement()`.

```
// Create a Statement object
try {
    stmt = mSQLcon.createStatement();
} catch (SQLException e) {
    System.out.println (e.getMessage());
}
```

Utilisez la méthode `Statement.executeUpdate()` pour soumettre un INSERT, un UPDATE ou un DELETE .

```
// Pass a query via the Statement object
int count = stmt.executeUpdate("DELETE from
    Customer WHERE ssn='999-55-6666'");
```

La méthode `Statement.executeUpdate()` renvoie un entier qui représente le nombre d'enregistrements affectés.

Utilisez la méthode `Statement.executeQuery()` pour soumettre l'instruction SQL à la base de données. Notez que JDBC transmet l'instruction SQL à la connexion de base de données sous-jacente sans modification. JDBC ne tente aucune interprétation des requêtes.

```
// Pass a query via the Statement object
ResultSet rs = stmt.executeQuery("SELECT DISTINCT *
    from Customer order by ssn");
```

La méthode `Statement.executeQuery()` renvoie un résultat de type `ResultSet` pour traitement ultérieur.

Instructions JDBC

requête préparée

En cas d'exécution répétitive des mêmes instructions SQL, l'utilisation d'un objet `PreparedStatement` s'avère intéressante. Une *requête préparée* est une instruction SQL précompilée qui est plus efficace qu'une répétition d'appels de la même instruction SQL. La classe `PreparedStatement` hérite de la classe `Statement` pour permettre le paramétrage des instructions JDBC. Le code suivant présente un exemple d'utilisation d'une instruction préformatée :

Exemple

```
public boolean prepStatement(Reservation obj){
    PreparedStatement prepStmnt = msqlConn.prepareStatement(
"UPDATE Flights SET numAvailFCSeats = ? WHERE flightNumber = ?" );
    prepStmnt.setInt(1,
        (Integer.parseInt(obj.numAvailFCSeats) - 1));
    prepStmnt.setLong(2, obj.FlightNo);
    int rowsUpdated = prepStmnt.executeUpdate();
    return (rowsUpdated > 0) ;
}
```

Les méthodes setXXX

Les méthodes `setXXX` de configuration des paramètres SQL IN doivent indiquer les types compatibles avec le type SQL de paramètre d'entrée défini. Ainsi, si un paramètre IN est du type SQL `Integer`, `setInt` doit être utilisé.



Méthodes setxxx

Table 2: Méthodes setXXX et types SQL

Méthode	Type(s) SQL
setASCIIStream	Utilise une chaîne ASCII pour générer un LONGVARCHAR
setBigDecimal	NUMERIC
setBinaryStream	LONGVARBINARY
setBoolean	BIT
setByte	TINYINT
setBytes	VARBINARY ou LONGVARBINARY (selon la taille par rapport aux limites de VARBINARY)
setDate	DATE
setDouble	DOUBLE
setFloat	FLOAT
setInt	INTEGER
setLong	BIGINT
setNull	NULL
setObject	L'objet Java défini est converti en type SQL cible avant d'être envoyé
setShort	SMALLINT
setString	VARCHAR ou LONGVARCHAR (selon la taille par rapport aux limites du pilote sur VARCHAR)
setTime	TIME
setTimestamp	TIMESTAMP
setUnicodeStream	UNICODE

Instructions JDBC

procédure stockée

Une *procédure stockée* permet l'exécution d'instructions non SQL dans la base de données. La classe `CallableStatement` hérite de la classe `PreparedStatement` qui fournit les méthodes de configuration des paramètres IN. Etant donné que la classe `PreparedStatement` hérite de la classe `Statement`, la méthode de récupération de résultats multiples par une procédure enregistrée est supportée par la méthode `Statement.getMoreResults`.

Ainsi, vous pourriez utiliser une instruction `CallableStatement` pour enregistrer une instruction SQL précompilée, vous permettant d'interroger une base de données contenant les informations sur la disponibilité des sièges pour un vol particulier.

```
String planeID = "727";
CallableStatement querySeats = msqlConn.prepareCall("{call
    return_seats(?, ?, ?, ?)}");
try {
    querySeats.setString(1, planeID);
    querySeats.registerOutParameter(2, java.sql.Type.INTEGER);
    querySeats.registerOutParameter(3, java.sql.Type.INTEGER);
    querySeats.registerOutParameter(4, java.sql.Type.INTEGER);
    querySeats.execute();
    int FCSeats = querySeats.getInt(2);
    int BCSeats = querySeats.getInt(3);
    int CCSeats = querySeats.getInt(4);
} catch (SQLException SQLEx) {
    System.out.println("Query failed");
    SQLEx.printStackTrace();
}
```

Avant d'exécuter un appel de procédure stockée, vous devez explicitement appeler `registerOutParameter` pour enregistrer le type `java.sql.Type` de tous les paramètres SQL OUT.



Instructions JDBC

Batch

```
Connection cnx = DriverManager.getConnection(url) ;
cnx.setAutoCommit(false) ;
Statement s = cnx.createStatement() ;
s.addBatch("DELETE FROM personnes WHERE nom='Dupond'") ;
s.addBatch("DELETE FROM personnes WHERE nom='Dupont'") ;
s.addBatch("INSERT INTO personnes VALUES ('tryphon', 'Tournesol')");
s.executeBatch();
....
cnx.commit() ;
```


Instructions JDBC

Récupération de résultats

Le résultat de l'exécution d'une instruction peut se présenter sous forme de table de données accessible via un objet `java.sql.ResultSet`. Cette table se compose d'une série de lignes et de colonnes. Les lignes sont récupérées dans l'ordre. Un objet `ResultSet` maintient un curseur sur la ligne de données courante et le positionne tout d'abord sur la première ligne. Le premier appel de l'instruction `next` définit la première ligne en tant que ligne courante, le second appel déplace le curseur sur la seconde ligne, etc.

A partir de la plateforme Java 2 les `ResultSets` offrent de nouveaux services :

- regroupement de résultats en blocs : `rs.fetchsize(25) ;`
- navigation dans les résultats : `rs.previous(), ...`

L'objet `ResultSet` fournit une série de méthodes `get` permettant d'accéder aux nombreuses valeurs de colonne de la ligne courante. Ces valeurs peuvent être récupérées à partir du nom de la colonne ou d'un indice. Il est généralement plus pratique d'utiliser un indice pour référencer une colonne. Les indices de colonne débutent à 1.



Instructions JDBC

Réception de résultats (suite)

```
while (rs.next()) {
    System.out.println ("Customer: " + rs.getString(2));
    System.out.println ("Id:          " + rs.getString(1));
    System.out.println ("");
}
```

Les diverses méthodes `getXXX` accèdent aux colonnes dans la table de résultats. Il est possible d'accéder aux colonnes d'une ligne dans n'importe quel ordre.

Nota: il est possible de découvrir dynamiquement des informations sur la table comme le nombre de champs dans un enregistrement, le type de chaque champ, etc. Ce type d'information est géré par l'objet `ResultSetMetaData` rendu par `getMetaData()` -service non accessible en `mSQL`-

Pour récupérer des données extraites de l'objet `ResultSet`, vous devez vous familiariser avec les colonnes retournées, ainsi qu'avec les types de données qu'elles contiennent. La table 2-3 établit une correspondance entre les types de données Java et SQL.

Méthodes get_{xxx}

Table 3: Méthodes get_{XXX} et type de données Java retourné

Méthode	Type de données Java retourné
getASCIIStream	java.io.InputStream
getBigDecimal	java.math.BigDecimal
getBinaryStream	java.io.InputStream
getBoolean	boolean
getByte	byte
getBytes	byte[]
getDate	java.sql.Date
getDouble	double
getFloat	float
getInt	int
getLong	long
getObject	Object
getShort	short
getString	java.lang.String
getTime	java.sql.Time
getTimestamp	java.sql.Timestamp
getUnicodeStream	java.io.InputStream de caractères Unicode



Correspondance des types de données SQL en Java

La table 2-3 présente les types Java standard pour la correspondance avec divers types SQL courants.

Table 4: Correspondance de types SQL en Java

Type SQL	Type Java
CHAR	String
VARCHAR	String
LONGVARCHAR	String (ou Stream)
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Utilisation de l'API JDBC

En créant une série générique d'interfaces permettant d'établir une connexion à tout produit de base de données grâce à l'API JDBC, vous n'êtes limité ni à une base de données spécifique, ni à une architecture particulière d'accès car plusieurs solutions peuvent être envisagées.

Types de conception des pilotes JDBC

La conception de l'architecture des accès base de données que vous choisirez d'implanter dépend en partie du pilote JDBC.

- **Conception en deux éléments** – Utilisant le langage Java vers des bibliothèques de méthodes natives ou Java vers un protocole de système de gestion de base de données (SGBD) natif (tous les codes en Java mais protocole spécifique)
- **Conception en trois éléments** – Utilisant tous les codes Java dans lesquels les appels JDBC sont convertis en protocole indépendant du système SGBD.
- **Conception en deux ou trois éléments** – Spécialement conçue pour fonctionner avec les pilotes de bases de données ODBC (connectivité ouverte aux bases de données Microsoft), dans lesquels les appels JDBC sont effectués par le biais d'un pilote ODBC (généralement une librairie spécifique à la plate-forme)

A ce jour, les principaux développements de pilotes de bases de données ont été réalisés par des constructeurs indépendants, non associés à une société de fourniture de bases de données. Cette situation est avantageuse pour le développeur car elle implique que les solutions ne sont généralement pas privées et que la concurrence régule les prix.





Contenu:

Cette annexe introduit l'API **Java Native Interface** qui permet d'étendre JAVA avec du code compilé écrit en C ou C++:

- Pourquoi réaliser du code natif?
- Les phases de génération : un exemple simple
- Les caractéristiques générales de JNI
- Exemples : emploi des types Java en C, accès aux attributs des objets JAVA, création d'instances.
- Le problème de l'intégrité des références aux objets JAVA.
- Le traitement des exceptions JAVA
- L'invocation de JAVA à l'intérieur d'un code C/C++.



Pourquoi réaliser du code natif?

Il y a des situations dans laquelle le code ne peut pas être complètement écrit en JAVA :

- Une grande quantité de code compilé existe déjà et fonctionne de manière satisfaisante. La fourniture d'une interface avec JAVA peut être plus intéressante qu'une réécriture complète.
- Une application doit utiliser des services non fournis par JAVA (et en particulier pour exploiter des spécificités de la plate-forme d'exécution. Exemple : accès à des cartes).
- Le système JAVA n'est pas assez rapide pour des applications critiques et la réalisation dans un code natif serait plus efficiente.

Il est possible d'implanter en JAVA des méthodes *natives* réalisées typiquement en C ou C++.

Une classe comprenant des méthodes natives ne peut pas être téléchargée au travers du réseau de manière standard: il faudrait que le serveur ait connaissance des spécificités de la plate-forme du client. De plus une telle classe ne peut faire appel aux services de sécurité de JAVA (en 1.1)

Bien entendu pour toute application JAVA qui s'appuie sur des composants natifs on doit réaliser un portage du code natif sur chaque plate-forme spécifique. De plus c'est un code potentiellement plus fragile puisque les contrôles (pointeurs, taille, etc.) et la récupération d'erreurs sont entièrement sous la responsabilité du programmeur.

Il existe diverses manières d'assurer cette liaison code compilé-Java. Depuis la version JAVA 1.1 le protocole JNI a été défini pour rendre cette adaptation plus facilement indépendante de la réalisation de la machine virtuelle sous-jacente.

D'autre part il est également possible d'exécuter du code JAVA au sein d'une application écrite en C/C++ en appelant directement la machine virtuelle (JAVA "enchassé" dans C).

un exemple : "Hello World" en C

résumé des phases :

Ecriture du code JAVA :

- Création d'une classe "HelloWorld" qui déclare une méthode (statique) native.

Création des binaires JAVA de référence :

- Compilation du code ci-dessus par `javac` .

Génération du fichier d'inclusion C/C++ :

- Ce fichier est généré par l'utilitaire `javah` . Il fournit une définition d'un en-tête de fonction C pour la réalisation de la méthode native `getGreetings()` définie dans la classe Java "HelloWorld".

Ecriture du code natif :

- Ecriture d'un fichier source C (".c") qui réalise en C le code de la méthode native. Ce code fait appel à des fonctions et des types prédéfinis de JNI.

Création d'une librairie dynamique:

- Utilisation du compilateur C pour générer une librairie dynamique à partir des fichiers .c et .h définis ci-dessus. (sous Windows une librairie dynamique est une DLL)

Exécution:

- Exécution du binaire JAVA (par `java`) avec chargement dynamique de la librairie.



un exemple : "Hello World" en C

Ecriture du code JAVA

Le code de l'exemple définit une classe JAVA nommée "HelloWorld" et faisant partie du package "hi".

```
package hi ;

class HelloWorld {

    static {
        System.loadLibrary("hello");
    }
    public static native String getGreetings();

    public static void main (String[] tArgs) {
        for (int ix = 0 ; ix < tArgs.length; ix++) {
            System.out.println(getGreetings() + tArgs[ix]) ;
        }
    } // main
}
```

Cette classe pourrait contenir également d'autres définitions plus classiques (champs, méthodes, etc.). On remarquera ici :

- La présence d'un bloc de code `static` exécuté au moment du chargement de la classe. A ce moment il provoque alors le chargement d'une bibliothèque dynamique contenant le code exécutable natif lié à la classe.
Le système utilise un moyen standard (mais spécifique à la plate-forme) pour faire correspondre le nom "hello" à un nom de bibliothèque ("libhello.so" sur Solaris, "hello.dll" sur Windows,...)
- La définition d'un en-tête de méthode **native**.
Une méthode marquée `native` ne dispose pas de corps. Comme pour une méthode `abstract` le reste de la "signature" de la méthode (arguments, résultat,...) doit être spécifié.
(ici la méthode est `static` mais on peut, bien sûr, créer des méthodes d'instance qui soient natives).

un exemple : "Hello World" en C

Création des binaires JAVA de référence

La classe ainsi définie se compile comme une autre classe :

```
javac -d . HelloWorld.java
```

(autre exemple sous UNIX : au lieu de "-d ." on peut faire par exemple "-d \$PROJECT/javaclasses")

Le binaire JAVA généré est exploité par les autres utilitaires employés dans la suite de ce processus.

Dans l'exemple on aura un fichier "HelloWorld.class" situé dans le sous-répertoire "hi" du répertoire ciblé par l'option "-d"



un exemple : "Hello World" en C

Génération du fichier d'inclusion C/C++

L'utilitaire javah va permettre de générer à partir du binaire JAVA un fichier d'inclusion C/C++ ".h". Ce fichier définit les prototypes des fonctions qui permettront de réaliser les méthodes natives de HelloWorld.

```
javah -d . -jni hi.HelloWorld
```

(autre exemple sous UNIX: javah -d \$PROJECT/csources)

On obtient ainsi un fichier nommé hi_HelloWorld.h :

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class hi_HelloWorld */

#ifdef _Included_hi_HelloWorld
#define _Included_hi_HelloWorld
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:      hi_HelloWorld
 * Method:     getGreetings
 * Signature:  ()Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_hi_HelloWorld_getGreetings
    (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endif
```

Des règles particulières régissent la génération du nom de fichier d'inclusion et des noms de fonctions réalisant des méthodes natives.

On notera que la fonction rend l'équivalent d'un type JAVA (jstring) et, bien qu'étant définie sans paramètres en JAVA, comporte deux paramètres en C. Le pointeur d'interface JNIEnv permet d'accéder aux objets JAVA, jclass référence la classe courante (on est ici dans une méthode statique: dans une méthode d'instance le paramètre de type jobject référencerait l'instance courante).

un exemple : "Hello World" en C

Ecriture du code natif

En reprenant les prototypes définis dans le fichier d'inclusion on peut définir un fichier source C : "hi_HelloWorldImp.c" :

```
#include <jni.h>
#include "hi_HelloWorld.h"

/*
 * Class:      hi_HelloWorld
 * Method:    getGreetings
 * Signature: ()Ljava/lang/String;
 * on a une methode statique et c'est la classe
 * qui est passée en paramètre
 */
JNIEXPORT jstring JNICALL Java_hi_HelloWorld_getGreetings
    (JNIEnv * env , jclass curclass) {

    return (*env)->NewStringUTF(env, "Hello ");
}
```

env nous fournit une fonction NewStringUTF qui nous permet de générer une chaîne JAVA à partir d'une chaîne C.

NOTA : en C++ les fonctions JNI sont "inline" et le code s'écrirait :

```
JNIEXPORT jstring JNICALL Java_hi_HelloWorld_getGreetings
    (JNIEnv * env , jclass curclass) {

    return env->NewStringUTF("Hello ");
}
```



un exemple : "Hello World" en C

Création d'une librairie dynamique

Exemple de génération sous UNIX (le ".h" est dans le répertoire courant)

```
#!/bin/sh
# changer DIR en fonction des besoins
DIR=/usr/local/java
cc -G -I$DIR/include -I$DIR/include/solaris \
    hi_HelloWorldImp.c -o libhello.so
```

Exemple de génération sous Windows avec le compilateur VisualC++4.0:

```
cl -Ic:\java\include -Ic:\java\include\win32 -LD
    hi_HelloWorldImp.c -Fehello.dll
```

un exemple : "Hello World" en C

Exécution

```
java hi.HelloWorld World underWorld
Hello World
Hello underWorld
```

Si, par contre, vous obtenez une exception ou un message indiquant que le système n'a pas pu charger la librairie dynamique il faut positionner correctement les chemins d'accès aux librairies dynamiques (LD_LIBRARY_PATH sous UNIX)



présentation de JNI

JNI est une API de programmation apparue à partir de la version 1.1 de JAVA. Il existait auparavant d'autres manières de réaliser des méthodes natives. Bien que ces autres APIs soient toujours accessibles elles présentent quelques inconvénients en particulier parce qu'elles accèdent aux champs des classes JAVA comme des membres de structures C (ce qui oblige à recompiler le code quand on change de machine virtuelle) ou parcequ'elles posent quelques problèmes aux glaneurs de mémoire (*garbage collector*).

Les fonctions de JNI sont adressables au travers d'un environnement (pointeur d'interface vers un tableau de fonctions) spécifique à un *thread*. C'est la machine virtuelle elle même qui passe la réalisation concrète de ce tableau de fonctions et on assure ainsi la compatibilité binaire des codes natifs quel que soit la machine virtuelle effective.

Les fonctions proposées permettent en particulier de :

- Créer, consulter, mettre à jour des objets JAVA, (et opérer sur leurs verrous). Opérer avec des types natifs JAVA.
- Appeler des méthodes JAVA
- Manipuler des exceptions
- Charger des classes et inspecter leur contenu

Le point le plus délicat dans ce partage de données entre C et JAVA et celui du glaneur de mémoire (*garbage collector*): il faut se protéger contre des déréréfencements d'objets ou contre des effets de compactage en mémoire (déplacements d'adresses provoqués par gc), mais il faut savoir aussi faciliter le travail du glaneur pour récupérer de la mémoire.

JNI: types, accès aux membres, création d'objets

Soit l'exemple de classe :

```
package hi ;
class Uni {
    static {
        System.loadLibrary("uni");
    }
    public String [] tb ;// champ "tb"
    public Uni(String[] arg) {
        tb = arg ;
    }// constructeur
    public native String [] getMess(int n, String mess);
    public static native Uni dup(Uni other);

    public String toString() {
        String res = super.toString() ;
        // pas efficient
        for (int ix = 0 ; ix < tb.length; ix ++) {
            res = res + '\n' + tb[ix] ;
        }
        return res ;
    }
    public static void main (String[] tArgs) {
        Uni you = new Uni(tArgs) ;
        System.out.println(you) ;
        String[] mess = you.getMess(tArgs.length,
                                   " Hello") ;
        for (int ix = 0 ; ix < mess.length; ix++) {
            System.out.println(mess[ix]) ;
        }
        Uni me = Uni.dup(you) ;
        System.out.println(me) ;
    }// main
}
```

Exemple d'utilisation :

```
java hi.Uni World
hi.Uni@1dce0764
World
Hello
hi.Uni@1dce077f
World
```



JNI: types, accès aux membres, création d'objets

La méthode native `getMess(int nb, String mess)` génère un tableau de chaînes contenant "nb" fois le même message "mess" :

```

/* Class:      hi_Uni
 * Method:     getMess
 * Signature:  (ILjava/lang/String;) [Ljava/lang/String;
 */
JNIEXPORT jobjectArray JNICALL
Java_hi_Uni_getMess (JNIEnv * env , jobject curInstance,
                    jint nb , jstring chaine) {
    /* quelle est la classe de String ? */
    jclass stringClass = (*env)->FindClass(env,
                                           "java/lang/String") ;
    /* un tableau de "nb" objet de type "stringClass"
     * chaque element est initialise a "chaine" */
    return (*env)->NewObjectArray(env,
                                   (jsize)nb, stringClass, chaine) ;
}

```

- La fonction `NewObjectArray` est une des fonctions de création d'objets JAVA. Elle doit connaître le type de ses composants (ici fourni par "stringClass").
L'initialisation de chaque membre d'un tel tableau se fait par l'accessoire `SetObjectArrayElement()` - mais ici on profite du paramètre d'initialisation par défaut-
- JNI fournit des types C prédéfinis pour représenter des types primitifs JAVA (`jint`) ou pour des types objets (`jobject`, `jstring`, ..)
- La fonction `FindClass` permet d'initialiser le bon paramètre désignant la classe "java.lang.String" (la notation utilise le séparateur "/"!).
Noter également la représentation de la signature de la fonction "getMess": `(ILjava/lang/String;)` indique un premier paramètre de type `int` (symbolisé par la lettre I) suivi d'un objet (lettre L+ type + ;).
De même `[Ljava/lang/String;` désigne un résultat qui est un tableau à une dimension (lettre L) contenant des chaînes.

JNI: types, accès aux membres, création d'objets

La méthode statique "dup" clone l'instance passée en paramètre :

```

/* Class:      hi_Uni; Method:    dup
 * Signature: (Lhi/Uni;)Lhi/Uni; */
JNIEXPORT jobject JNICALL Java_hi_Uni_dup (JNIEnv * env,
                                           jclass curClass , jobject other) {
    jfieldID idTb ; jobjectArray tb ;
    jmethodID idConstr ;
    /* en fait inutile puisque c'est curClass !*/
    jclass uniClass = (*env)->GetObjectClass(env, other) ;
    if(! (idTb = (*env)->GetFieldID (env,uniClass,
                                     "tb", "[Ljava/lang/String;")))
        return NULL ;
    tb = (jobjectArray) (*env)->GetObjectField(env,
        other,idTb) ;

    /* on initialise un nouvel objet */
    if(! (idConstr = (*env)->GetMethodID(env, curClass,
        "<init>", "([Ljava/lang/String;)V")))
        return NULL ;
    return (*env)->NewObject (env,
        curClass,idConstr,tb) ;
}

```

- La récupération du champ "tb" (de type tableau de chaîne) sur l'instance passée en paramètre se fait par la fonction `GetObjectField`. On a besoin de la classe de l'instance consultée et de l'identificateur du champ qui est calculé par `GetFieldID`.
- De la même manière l'appel d'une méthode nécessite une classe et un identificateur de méthode calculé par `GetMethodID`. Ici ce n'est pas une méthode qui est appelée mais un constructeur et l'identifiant est calculé de manière particulière ("`<init>`"), le type indique un paramètre de type tableau de chaîne et un "résultat" qui est `void` (lettre `V`).

JNI fournit ainsi des accesseurs à des champs (`Get<static><type>Field`, `Set<static><type>Field`) et des moyens d'appeler des méthodes (`Call<statut><type>Method` : exemple `CallStaticBooleanMethod`). Il existe, en plus, des méthodes spécifiques aux tableaux et aux Strings



références sur des objets JAVA:

Le passage du glaneur de mémoire sur des objets JAVA pourrait avoir pour effet de rendre leur référence invalide ou de les déplacer en mémoire. Les références d'objet JAVA transmises dans les transitions vers le code natif sont protégées contre les invalidations (elles redeviennent récupérables à la sortie du code natif).

Toutefois JNI autorise le programmeur à explicitement rendre une référence locale récupérable. Inversement il peut aussi se livrer à des opérations de "punaisage" (*pinning*) lorsque, pour des raisons de performances, il veut pouvoir accéder directement à une zone mémoire protégée :

```
const char * str = (*env)->GetStringUTFChars(env,
                                             javaString,0) ;
.... /* opérations sur "str" */
(*env)->ReleaseStringUTFChars(env, javaString, str) ;
```

Des techniques analogues existent pour les tableaux de scalaires primitifs (int, float, etc.). Bien entendu il est essentiel que le programmeur C libère ensuite la mémoire ainsi gelée.

Si on veut éviter de bloquer entièrement un tableau alors qu'on veut opérer sur une portion de ce tableau , on peut utiliser des fonctions comme :

```
void GetIntArrayRegion(JNIenv* env, jintArray tableau,
                      jsize debut, jsize taille, jint * buffer) ;
void SetIntArrayRegion(....
```

Le programmeur a aussi la possibilité de créer des références globales sur des objets JAVA. De telles références ne sont pas récupérables par le glaneur à la sortie du code natif, elles peuvent être utilisées par plusieurs fonctions implantant des méthodes natives. Ici aussi il est de la responsabilité du programmeur de libérer ces références globales.

exceptions

JNI permet de déclencher une exception quelconque ou de récupérer une exception JAVA provoquée par un appel à une fonction JNI. Une exception JAVA non récupérée par le code natif sera retransmise à la machine virtuelle .

```
....
jthrowable exc;
(*env)->CallVoidMethod(env, instance , methodID) ;
/* quelque chose s'est-il produit? */
exc = (*env)->ExceptionOccurred(env);
if (exc) {
    jclass NouvelleException ;
    ... /* diagnostic */
    /* on fait le ménage */
    (*env)->ExceptionClear(env) ;
    /* et on fait du neuf ! */
    NouvelleException = (*env)->FindClass(env,
        "java/lang/IllegalArgumentException") ;
    (*env)->ThrowNew(env,NouvelleException, message);
}
...
```



invocation de JAVA dans du C

```

/*lanceur.c
 * usage : lanceur <classe_JAVA_en_format_> <args> */
JavaVM *jvm ;
JNIEnv * env;
JDK1_1InitArgs vm_args ; /* 1.2 : JavaVMInitArgs */

main(argc, argv) int argc; char ** argv; {
    int cnt ; jint res ; jclass mainclass ;
    jmethodID methodID; jobjectArray jargs ;
    /* ici controles de lancement a faire */
    ....
    /* initialisation des champs de vm_args ATTENTION*/
    vm_args.version = 0x00010001 ; /* CHANGE en 1.2 ! */
    /* appel obsolete en JAVA 1.2 */
    res = JNI_GetDefaultJavaVMInitArgs(&vm_args) ;
    /* APPEL OBSOLETE (non portable). CHANGE en 1.2 */
    vm_args.classpath = getenv("CLASSPATH");
    if (0 > JNI_CreateJavaVM(&jvm, &env, &vm_args))exit(1) ;

    if (!(mainclass= (*env)->FindClass(env,argv[1])))
        exit(1);
    if(!(methodID= (*env)->GetStaticMethodID(env,mainclass,
        "main", "([Ljava/lang/String;)V"))exit(2);
    ....
    jargs = (*env)->NewObjectArray(env, (jsize) (argc-2),
        (*env)->FindClass(env, "java/lang/String"), NULL) ;
    for (cnt = 0 ; cnt < (argc - 2) ; cnt++) {
        jobject stringObj = (*env)->NewStringUTF(env,
            argv[cnt+2]);
        (*env)->SetObjectArrayElement (env,
            jargs, (jsize)cnt, stringObj);
    }
    (*env)->CallStaticVoidMethod(env,
        mainclass, methodID, jargs) ;
    .....
    (*jvm)->DestroyJavaVM(jvm) ;
}

```

Un tel code doit être lié à la librairie binaire JAVA (`libjava.so` sous UNIX).

La version de JAVA 2 introduit quelques modifications (voir documentation).



Points essentiels

Les collections permettent de stocker des objets hétérogènes.

- Avant la version Java 2 il existait un certain nombre de classes rendant des services de stockage d'objets : Vector, Hashtable, etc.
- Java 2 définit une architecture plus générale qui permet de mieux catégoriser ces services. On dispose d'interfaces associées à des sémantiques (par exemple : ensemble, dictionnaire, etc.), de classes de base réalisant ces services et d'algorithmes généraux opérant sur des collections (tris, recherche, etc.).



généralités

Une “Collection” est un objet servant à regrouper d’autres objets. Les collections sont utilisées comme moyen de stockage (un tableau d’objets `Object[]` est en quelque sorte une collection de bas niveau). Ce stockage est conçu aussi pour faciliter des manipulations comme la recherche d’un objet particulier.

Dans les versions de Java antérieures à la version 2 le package `java.util` proposait les services d’un petit nombre de classes : `Vector` , `Stack`, `Dictionary`, `Hashtable`, `Properties`.

Vector (java 1.1)

`Vector` permet de disposer d'un tableau extensible d'objets : la taille est ajustée automatiquement en fonction des besoins.

- l C'est une collection **ordonnée** :
On peut accéder aux objets par leur index (l'exception `ArrayIndexOutOfBoundsException` est générée en cas d'erreur)
- l C'est une collection **extensible** :
Les nouveaux éléments sont automatiquement rajoutés après le dernier élément du vecteur.
- l C'est une collection **contiguë** :
La suppression d'un élément entraîne la compaction du vecteur (on ne laisse pas de "trou"). De même l'insertion d'un élément à un index donné peut donner lieu à un déplacement des éléments situés après cet index.

Les principaux services d'un vecteur concernent :

- l La gestion de la capacité (elle est normalement automatique mais le programmeur peut intervenir pour des raisons d'optimisation)
- l La recherche d'objets :
 - l recherche par index
 - l recherche d'un objet particulier (par sa référence)
 - l parcours de tous les objets
- l La copie de la collection
- l La modification du vecteur (accès par index ou accès par valeur)

Un `Vector` peut facilement être utilisé pour réaliser des piles (FIFO, LIFO -voir `Stack`-).



Vector (java 1.1).

Vecteurs : optimisations de la capacité

La *capacité* d'un vecteur exprime le nombre d'éléments qu'il peut recevoir sans avoir à effectuer d'opération d'extension (opération coûteuse).

Cette information est donnée par la méthode `capacity()` qui est à distinguer de la méthode `size()` qui donne le nombre d'éléments effectivement présents dans le vecteur.

- I Le constructeur `Vector()` réserve un nombre fixe d'emplacements (actuellement 10) et double la capacité chaque fois qu'il doit s'étendre
- I Le constructeur `Vector(capacitéInitiale)` permet de fixer la capacité initiale (l'extension double la capacité).
- I Le constructeur `Vector(capacité, incrément)` fixe la capacité initiale et le nombre d'emplacements qui seront préparés au cours d'une extension (un incrément inférieur ou égal à zéro permet de déterminer une extension par doublement de capacité)
- I La méthode `ensureCapacity(tailleMinimum)` permet de déclencher éventuellement l'extension pour garantir une capacité minimum.

Vector (java 1.1)..

Vecteurs : recherche d'objets

I par **index** :

- I `elementAt (index)` fournit l'élément situé à l'index demandé. Cette méthode est susceptible de déclencher l'exception `ArrayIndexOutOfBoundsException`.
- I Les méthodes `firstElement ()` et `lastElement ()` peuvent provoquer l'exception `NoSuchElementException`.

I par **référence** :

- I Les méthodes `indexOf(objet)`, `indexOf(objet, indexDépart)`, `lastIndexOf(objet),...` rendent l'index de l'objet dont la référence est passée en paramètre (ou -1 si l'objet n'est pas trouvé). Comme un objet peut être référencé plusieurs fois dans un vecteur il existe plusieurs méthodes de recherche.
- I `contains (objet)` indique si l'objet est présent dans le vecteur.

I **opérations globales** :

- I `elements ()` permet d'initialiser un parcours (voir `Enumeration`)
- I `copyInto (Object [])` permet de recopier les éléments du vecteur dans un tableau d'objets; l'ordre des objets est conservé. Le tableau doit avoir une taille suffisante pour contenir toutes les références stockées dans le vecteur.
- I `clone ()` fournit un autre vecteur qui a les mêmes caractéristiques que le vecteur courant et qui référence les mêmes objets.



Vector (java 1.1)...

Vecteurs : modifications

l par index :

- l `insertElementAt(index)`, `removeElementAt(index)` sont susceptibles de provoquer une `ArrayIndexOutOfBoundsException`. Une insertion comme une destruction provoquent un glissement des index.
- l `addElement(objet)` rajoute un élément en fin de vecteur
- l `setSize(taille)` peut avoir pour effet soit de détruire des références (si on réduit la taille) soit de créer des références à `null` (si on aggrandit la taille).

l par valeur :

- l `removeElement(objet)` détruit la première référence rencontrée de l'objet passé en paramètre. Elle rend `false` si cette référence n'est pas trouvée

Hashtable (java 1.1)

Hashtable extends Dictionary réalise un dictionnaire dans lequel des "valeurs" peuvent être recherchées à l'aide de "clefs".

Les clefs et les valeurs peuvent être n'importe quel objet (non null).

Les performances des accès dépendent de plusieurs facteurs :

- 1 La taille initiale de la table de Hash (voir constructeurs)
- 1 La manière dont cette table est amenée à s'aggrandir (voir également les constructeurs)
- 1 La pertinence des clefs de hash qui sont calculées à partir de la méthode `hashCode()` de l'objet choisi comme clef d'accès. La classe de cet objet doit aussi réaliser la méthode `equals(autreObjet)` de manière adaptée.

Les principaux services d'une Hashtable concernent :

- 1 La gestion de la configuration (optimisations)
- 1 La recherche d'objets (la recherche se fait essentiellement par clef, mais la recherche par valeur est possible -bien que non performante-)
- 1 la modification du contenu (enregistrement et suppression d'une nouvelle paire clef/valeur)



Hashtable (java 1.1).

Hashtable : recherche d'objets

I Par clef :

- I** `Object get(clef)` rend l'objet associé à "clef" (null si la clef n'existe pas)
- I** `boolean containsKey(clef)` indique si la clef est présente dans la table

I Par valeur :

- I** `boolean contains(Object valeur)` indique si l'objet "valeur" est référencé dans la table (accès peu performant!)

I opérations globales :

- I** Enumeration `keys()` permet d'initialiser un parcours des clefs présentes dans la table
- I** Enumeration `elements()` permet d'initialiser un parcours des valeurs présentes dans la table.

Hashtable : modifications

- I** `put(clef, valeur)` permet d'insérer une nouvelle paire clef/valeur dans la table. Si une association ayant même clef existait auparavant l'ancienne "valeur" est retournée (sinon résultat null)
- I** `remove(clef)` supprime une association clef/valeur de la table. Ici aussi l'ancienne valeur est retournée (si la clef n'existe pas la méthode ne fait rien).

Properties

Properties extends Hashtable permet de réaliser une "liste de propriétés" c'est à dire un ensemble de champs ayant un nom (une chaîne de caractères) et une valeur exprimée de manière standard sous forme de chaîne.

L'avantage d'une liste de propriétés est de permettre de réaliser une structure qui contient des champs variables : la présence d'un champ donné n'est pas obligatoire et on peut étendre dynamiquement cette liste.

On représente de cette manière :

- I Des informations venues du système ou de ressources locales (toutes ne sont pas présentes ou toutes ne sont pas accessibles pour des raisons de sécurité)
- I Des informations présentant de nombreuses variantes qui amèneraient à définir des classes avec un très grand nombre de champs (dont la plupart resteraient inutilisés pour une instance donnée)
- I Des informations que l'on trouve pratique de représenter et d'initialiser avec des chaînes de caractères facilement "lisibles".
Exemple initialisation d'une instance de GridBagConstraints :

```
gridx=3  
gridy=5  
gridwidth=2  
anchor=WEST  
fill=VERTICAL  
.....
```



Properties.

Properties : initialisations

Lors de la construction d'une liste de propriétés on peut demander de disposer d'une liste de valeurs par défaut (une autre liste de propriétés préalablement construite et initialisée).

Les méthodes `load(InputStream)` ou `store(OutputStream, commentaire)` permettent d'initialiser les valeurs depuis une ressource (fichier, flot de communication,...) ou de sauvegarder ces valeurs dans une ressource (le champ "commentaire: s'inscrit en tête du fichier -sous forme de commentaire-)

Le système fournit des propriétés qui peuvent être consultées par `System.getProperties()`. Ces propriétés sont calculées par le système (`user.name`, `user.home`,...), sont initialisées par des ressources internes à Java (`java.vendor`,...) ou extraites de paramètres de lancement.
exemple (sous UNIX) :

```
java -Dmyvar=$myvar -Dautrevar=$autrevar package.Maclasse
```

Note – Attention : pour des raisons de portabilité il n'y a pas d'autre méthode de récupération de l'environnement (`System.getenv()` est obsolete)

Properties..

Properties : méthodes spécifiques

```
String getProperty(prop) //retourne la propriété demandée

String getProperty(prop, défaut)
    //retourne "défaut" (au lieu de "null")
    // si la propriété n'existe pas

Enumeration propertyNames()
    // liste des propriétés accessibles

String System.getProperty(prop)
    // SecurityException possible

boolean Boolean.getBoolean(prop)
    // recherche propriété Système avec valeur "true"
```

Caractéristiques d'une recherche de propriété :

- I** Le résultat est une chaîne (null si propriété inexistante)
- I** Si la propriété n'est pas trouvée on peut la rechercher dans une liste par défaut (ex: `propertyNames()` explore récursivement les listes par défaut).
- I** on peut fixer une valeur par défaut si la propriété n'existe pas.

Dans le cas d'une recherche d'une propriété système on peut déclencher un exception de sécurité si la propriété existe et si la politique de sécurité en empêche l'accès (ex: `user.home` pour une applet sans droits particuliers).



Enumeration (java 1.1)

Interface de parcours générique sur des collections :

```
Enumeration enum ;
... // initialisations par ex.
    // collection.elements()

while (enum.hasMoreElements ()) {
    Object obj = enum.nextElement ();
    ...
}
```

Permet de réaliser des parcours sur tous les éléments d'une collection d'une manière indépendante de la nature même de cette collection. On obtient éventuellement un découplage : la nature réelle de la collection peut évoluer sans qu'on ait à modifier les programmes qui la parcourent.

Collections en plateforme Java 2

Le package `java.util` a été profondément transformé et propose une architecture unifiée pour servir de cadre à des réalisations spécifiques.

Les nouvelles classes (parfois très proches des anciennes) ont des caractéristiques minimum communes et s'inscrivent chacune dans une hiérarchie bien précise des sémantiques .

Listes

- **ArrayList** est très proche de `Vector`: c'est un "tableau extensible" qui implante le contrat défini par l'interface **List** . On retrouve des caractéristiques de `Vector` (paramètres d'initialisation), mais la réalisation n'est pas synchronisée par défaut. Les services d'accès aux membres :
 - `get (index)` , `indexOf(Object)` , `contains (Object)` : permettent de rechercher un objet
 - `add(index, Object)` , `add (Object)` , `set(index, Object)` , `remove (index)` : modifient la liste (`set` remplace l'élément courant).

Ces méthodes font partie de l'interface **List** qui exprime une séquence d'objets dans laquelle l'accès positionnel est privilégié. Par souci d'harmonisation `Vector` a été doté des mêmes méthodes.

- **LinkedList** est galemment une `List` basée sur une réalisation de liste chaînée.
- **ListIterator** étend **Iterator** (qui lui même remplace `Enumeration`) en permettant un parcours d'une collection avec une sémantique d'ordre positionnel (par ex. retour en arrière, ajout d'un élément, etc.)



Collections : dictionnaires, ensembles

L'interface **Map** définit un comportement de dictionnaire : les valeurs stockées sont associées à une clef qui permet de les rechercher rapidement. Le contrat d'interface définit en particulier :

- les méthodes de modification : `put(Object clef, Object val); Object remove(clef) ;`
- les méthodes de recherche : `Object get(clef) ;`
- les méthodes permettant d'obtenir l'ensemble du contenu : `Collection values(); Set keySet() ;`

La classe de réalisation **HashMap** est très proche de `Hashtable` (sans synchronisation).

La classe **HashSet** utilise une table de hash pour implanter un comportement de **Set**. Un Set est un ensemble : toute valeur contenue n'existe qu'une seule fois -y compris null-. Le contrat d'interface définit en particulier :

- modifications : `boolean add(Object) (rend faux si l'objet existe déjà); remove(Object) ;`
- recherche : `contains(Object) ;`
- parcours: `Iterator iterator() ;`

La classe **TreeSet** implante un Set particulier : **SortedSet** (ensemble trié). La réalisation s'appuie sur la classe **TreeMap** (qui implante l'interface **SortedMap**). La réalisation effective de ces arbres balancés suppose que l'on ait le moyen de comparer deux éléments quelconques contenus dans la collection.

Ordre “naturel”, comparateurs

Tout mécanisme interne aux collections s’appuyant sur une relation d’ordre doit pouvoir s’appuyer sur un des dispositifs suivants:

- Un “ordre naturel” :
tous les objets de la collection doivent réaliser l’interface `java.lang.Comparable` c’est à dire implanter ;
`public int compareTo(Object autreobjet) ;`

la méthode devant rendre 0 si les deux objets sont considérés comme “égaux”, un nombre négatif si l’objet courant est plus petit que l’autre et un nombre positif dans le cas contraire.
En cas d’incompatibilité la comparaison doit générer une `ClassCastException`.
- Un objet “comparateur” :
capable de comparer deux objets quelconques dans la collection et implantant l’interface `java.util.Comparator` avec la méthode
`public int compare(Object lun, Object lautre) ;`

les conditions de comparaison étant les mêmes que pour `Comparable`.

Ces mécanismes sont utilisés par les collections triées et par les méthodes de tri comme `Collections.sort(List)` ou `Arrays.sort(Object[])`.

Si on utilise un “objet comparateur” celui-ci est passé en paramètre au constructeur de la collection triée ou en paramètre à la méthode statique de tri.

```
Arrays.sort(tableClients, new Comparator() {
    public int compare(Object lun, Object lautre) {
        return ((Client)lun).getNom().compareToIgnoreCase(
            ((Client)lautre).getNom()) ;
    }
}) ;
```



Classes de service : Collections , Arrays

La classe **Collections** (et la classe **Arrays**) sont des classes de service offrant des méthodes statiques opérant sur des Collections ou des tableaux.

- mise en ordre :
Arrays : `sort`, `binarySearch`
Collections : `sort`, `binarySearch`, `min`, `max`,
`reverseOrder`, `shuffle`;
- copie :
Arrays : `fill`, `asList`
Collections : `fill(list, Object)`, `copy(List, List)`,
`Set singleton(Object)`
- générations de collections ayant des propriétés particulières :
`unmodifiableXXXX()`, `synchronizedXXXX()` (exemple :
`List synchronizedList(List)`).

Attention : seuls les accès élémentaires à ces objets sont synchronisés, une boucle de parcours de ces collections par un `Iterator` doit être explicitement synchronisé sur l'objet cible.

```
Collection col = Collections.synchronizedCollection(maColl) ;
synchronized (col) {
    Iterator iter = col.iterator() ;
    while( iter.hasNext()) {
        quelqueChose(iter.next()) ;
    }
}
```

Itérateurs

Iterator permet un parcours abstrait d'une collection, les principes sont plus sophistiqués que ceux de `Enumeration` :

```
public interface Iterator {
    boolean hasNext() ;
    Object next() ;
    void remove() ;//facultatif!!!!
}
```

`remove()` retire le dernier élément rendu par `next()`. C'est le seul moyen de modifier de manière sûre une collection pendant le parcours d'un itérateur.

De la même manière **ListIterator** permet d'insérer ou de remplacer un élément en cours de parcours.

Ces modifications peuvent déclencher diverses exceptions :

- `UnsupportedOperationException` : la collection sous jacente ne fournit pas le service demandé
- `IllegalStateException` : par exemple deux demandes de `remove()` sur un même `next()`.
- `ConcurrentModificationException` : sur certaines collections (en fait sur la plupart des implantations standard) les itérateurs sont capables de détecter des modifications opérées éventuellement par un autre `Thread`.





Contenu :

Cette annexe donne un aperçu sur les composants AWT courants et sur leur manipulation.



Les composants :

Button

C'est un composant d'interface utilisateur de base de type "appuyer pour activer". Il peut être construit avec une étiquette de texte précisant son rôle .

```
Button b = new Button("Sample");  
add(b);  
b.addActionListener(this);
```

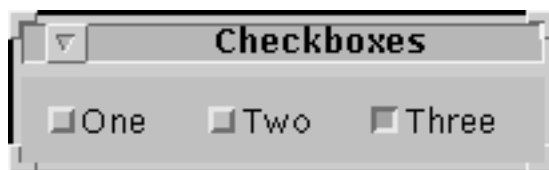


L'interface `ActionListener` doit pouvoir traiter un clic d'un bouton de souris. La méthode `getActionCommand()` de l'événement action (`ActionEvent`) activé lorsqu'on appuie sur le bouton rend par défaut la chaîne de l'étiquette.

Checkbox

La case à cocher fournit un dispositif d'entrée "actif/inactif" accompagné d'une étiquette de texte.

```
Checkbox one = new Checkbox("One", false);
Checkbox two = new Checkbox("Two", false);
Checkbox three = new Checkbox("Three", true);
add(one);
add(two);
add(three);
one.addItemListener(new Handler());
two.addItemListener(new Handler());
three.addItemListener(new Handler());
```



La sélection ou désélection d'une case à cocher est notifiée à la réalisation de l'interface `ItemListener`. Pour détecter une opération de sélection ou de désélection, il faut utiliser la méthode `getStateChange()` sur l'objet `ItemEvent`. Cette méthode renvoie l'une des constantes `ItemEvent.DESELECTED` ou `ItemEvent.SELECTED`, selon le cas. La méthode `getItem()` renvoie un objet de type chaîne (`String`) qui représente la chaîne de l'étiquette de la case à cocher considérée.

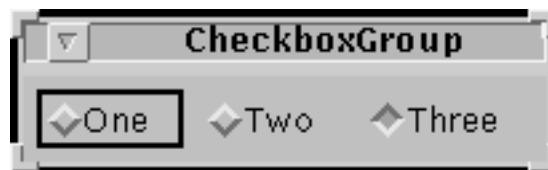
```
class Handler implements ItemListener {
    public void itemStateChanged(ItemEvent ev) {
        String state = "deselected";
        if (ev.getStateChange() == ItemEvent.SELECTED) {
            state = "selected";
        }
        System.out.println(ev.getItem() + " " + state);
    }
}
```



CheckboxGroup

On peut créer des cases à cocher à l'aide d'un constructeur spécial qui utilise un argument supplémentaire `CheckboxGroup`. Si on procède ainsi, l'aspect des cases à cocher est modifié et toutes les cases à cocher liées au même groupe adoptent un comportement de "bouton radio".

```
CheckboxGroup cbg = new CheckboxGroup();  
Checkbox one = new Checkbox("One", cbg, false);  
Checkbox two = new Checkbox("Two", cbg, false);  
Checkbox three = new Checkbox("Three", cbg, true);  
add(one);  
add(two);  
add(three);
```



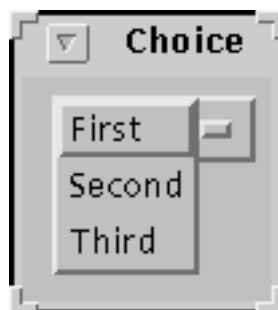
Choice

Le composant Choice fournit un outil simple de saisie de type "sélectionner un élément dans cette liste".

```
Choice c = new Choice();  
c.addItem("First");  
c.addItem("Second");  
c.addItem("Third");  
c.addItemListener(. . .);
```



Lorsqu'un composant Choice est activé il affiche la liste des éléments qui lui ont été ajoutés. Notez que les éléments ajoutés sont des objets de type chaîne (String).



L'interface `ItemListener` sert à observer les modifications de ce choix. Les détails sont les mêmes que pour la case à cocher. La méthode `getSelectedIndex()` de `Choice` permet de connaître l'index sélectionné.



List

Une liste permet de présenter à l'utilisateur des options de texte affichées dans une zone où plusieurs éléments peuvent être visualisés simultanément. Il est possible de naviguer dans la liste et d'y sélectionner un ou plusieurs éléments simultanément (mode de sélection simple ou multiple).

```
List l = new List(4, true);
```



L'argument numérique transmis au constructeur définit le nombre d'items visibles. L'argument booléen indique si la liste doit permettre à l'utilisateur d'effectuer des sélections multiples.



Un `ActionEvent`, géré par l'intermédiaire de l'interface `ActionListener`, est généré par la liste dans les modes de sélection simple et multiple. Les éléments sont sélectionnés dans la liste conformément aux conventions de la plate-forme. Pour un environnement Unix/Motif, cela signifie qu'un simple clic met en valeur une entrée dans la liste, mais qu'un double-clic déclenche l'action correspondante.

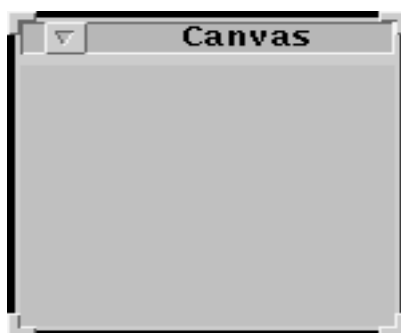
Récupération: voir méthodes `getSelectedObjects()`,
`getSelectedItems()`

Canvas

Un canvas fournit un espace vide (arrière-plan coloré). Sa taille par défaut étant zéro par zéro, on doit généralement s'assurer que le gestionnaire de disposition lui affectera une taille non nulle.

Cet espace peut être utilisé pour dessiner, recevoir du texte ou des saisies en provenance du clavier ou de la souris.

Le canvas est généralement utilisé tel quel pour fournir un espace de dessin général.



Le canvas peut écouter tous les événements applicables à un composant général. On peut, en particulier, lui associer des objets `KeyListener`, `MouseMotionListener` ou `MouseListener` pour lui permettre de répondre d'une façon ou d'une autre à une interaction utilisateur.



Label

Un label affiche une seule ligne de texte. Le programme peut modifier le texte. Aucune bordure ou autre décoration particulière n'est utilisée pour délimiter un label.

```
Label lab = new Label("Hello");  
add(lab);
```



En général, on ne s'attend pas à ce que les `Labels` traitent des événements, pourtant ils effectuent cette opération de la même façon qu'un `canvas`. Dans ce cas, on ne peut capter les activations de touches de façon fiable qu'en faisant appel à `requestFocus()`.

TextArea

La zone de texte est un dispositif de saisie de texte multi-lignes, multi-colonnes. On peut le rendre non éditable par l'intermédiaire de la méthode `setEditable(boolean)`. Il affiche des barres de défilement horizontales et verticales.

```
TextArea t = new TextArea("Hello!", 4, 30);  
add(t);
```



On peut ajouter des veilleurs d'événements de divers type dans une zone de texte.

Le texte étant multi-lignes, le fait d'appuyer sur <Entrée> place seulement un autre caractère dans la mémoire tampon. Si on a besoin de savoir à quel moment une saisie est terminée, on peut placer un bouton de validation à côté d'une zone de texte pour permettre à l'utilisateur de fournir cette information.

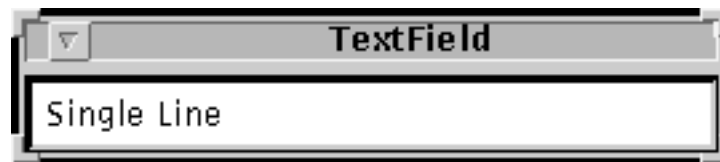
Un veilleur `KeyListener` permet de traiter chaque caractère entré en association avec la méthode `getKeyChar()`, `getKeyCode()` de la classe `KeyEvent`.



TextField

Le `TextField` est un dispositif de saisie de texte sur une seule ligne.

```
TextField f = new TextField("Single line", 30);  
add(f);
```



Du fait qu'une seule ligne est possible, un écouteur d'action (`ActionListener`) peut être informé, via `actionPerformed()`, lorsque la touche <Entrée> ou <Retour> est activée.

Comme la zone de texte, le champ texte peut être en lecture seule. Il n'affiche pas de barres de défilement dans l'une ou l'autre direction mais permet, si besoin est, un défilement de gauche à droite d'un texte trop long.

TextComponent

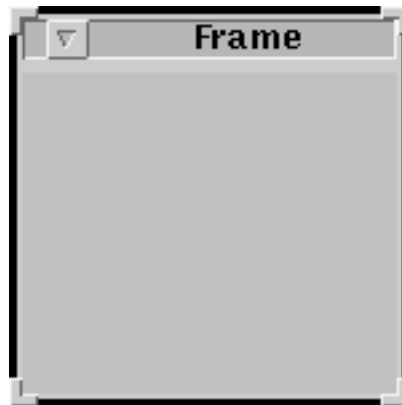
La classe `TextComponent` dont dérivent `TextField` et `TextArea` fournit un grand nombre de méthodes.

On a vu que les constructeurs des classes `TextArea` et `TextField` permettent de définir un nombre de colonnes pour l'affichage. Le nombre de colonnes est interprété en fonction de la largeur moyenne des caractères dans la police utilisée. Le nombre de caractères effectivement affichés peut varier radicalement en cas d'utilisation d'une police à chasse proportionnelle.

Frame

C'est la fenêtre générale de "plus haut niveau". Elle possède des attributs tels que : barre de titre et zones de contrôle du redimensionnement.

```
Frame f = new Frame("Frame");
```



La taille d'un `Frame` peut être définie à l'aide de la méthode `setSize()` ou avec la méthode `pack()`. Dans ce cas le gestionnaire de disposition calcule une taille englobant tous les composants du `Frame` et définit la taille de ce dernier en conséquence.

Les événements du `Frame` peuvent être surveillés à l'aide de tous les gestionnaires d'événements applicables aux composants généraux. `WindowListener` peut être utilisé pour réagir, via la méthode `windowClosing()`, lorsque le bouton `Quit` a été activé dans le menu du gestionnaire de fenêtres.

Il n'est pas conseillé d'écouter des événements clavier directement à partir d'un `Frame`. Bien que la technique décrite pour les composants de type `Canvas` et `Label`, à savoir l'appel de `requestFocus()`, fonctionne parfois, elle n'est pas fiable. Si on a besoin de suivre des événements clavier, il est plutôt recommandé d'ajouter au `Frame` un `Canvas`, `Panel`, etc., et d'associer le gestionnaire d'événement à ce dernier.



Panel

C'est le conteneur de base. Il ne peut pas être utilisé de façon isolée comme les Frames, les fenêtres et les boîtes de dialogue.

```
Panel p = new Panel();
```

Les Panels peuvent gérer les événements (rappel : le focus clavier doit être demandé explicitement).

Dialog

Un `Dialog` est une fenêtre qui diffère toutefois d'un `Frame` :

- elle est destinée à afficher des messages fugitifs
- elle peut être modale: elle recevra systématiquement toutes les saisies jusqu'à fermeture.
- elle ne peut-être supprimée ou icônifiée par les boutons du gestionnaire de fenêtre, on lui associe habituellement un bouton de validation.



```
Dialog d = new Dialog(f, "Dialog", false);  
d.add(new Label("Hello, I'm a Dialog"),  
        BorderLayout.CENTER);  
d.pack();
```



Dialog

Un dialog dépend d'une Frame : cette Frame apparait comme premier argument dans les constructeurs de la classe Dialog.

Les boîtes de dialogue ne sont pas visibles lors de leur création. Elles s'affichent plutôt en réponse à une autre action au sein de l'interface utilisateur, comme le fait d'appuyer sur un bouton.

```
public void actionPerformed(ActionEvent ev) {  
    d.setVisible(true);  
}
```



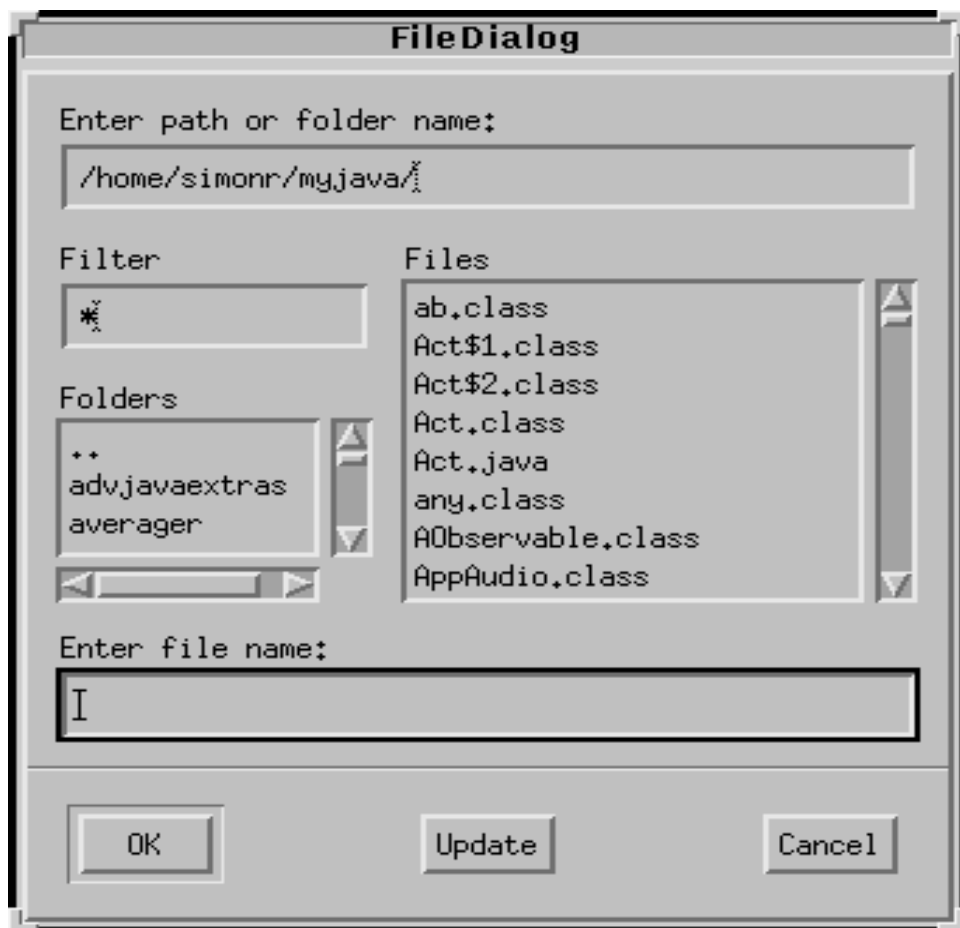
Il est recommandé de considérer une boîte de dialogue comme un dispositif réutilisable. Ainsi, vous ne devez pas détruire l'objet individuel lorsqu'il est effacé de l'écran, mais le conserver pour une réutilisation ultérieure.

Pour masquer une boîte de dialogue, appelez `setVisible(false)`. Cette opération s'effectue généralement en ajoutant un `WindowListener`, et en attendant que la méthode `windowClosing()` soit appelée dans ce gestionnaire d'événement.

FileDialog

C'est une implantation d'un dispositif de sélection de fichier. Elle comporte sa propre fenêtre autonome et permet à l'utilisateur de parcourir le système de fichiers et de sélectionner un fichier spécifique pour des opérations ultérieures.

```
FileDialog d = new FileDialog(f, "FileDialog");  
d.setVisible(true);  
String fname = d.getFile();
```



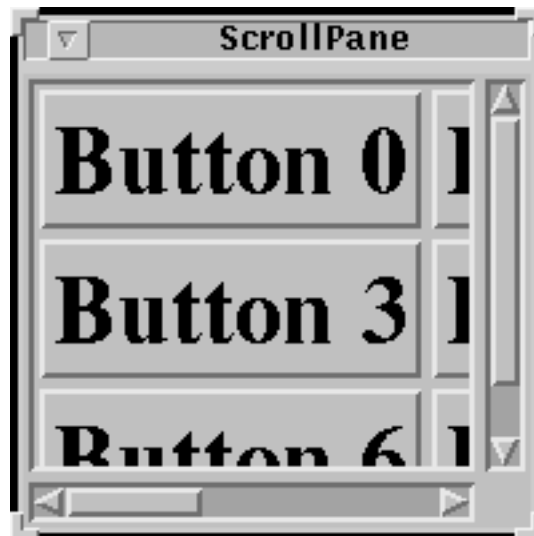
En général, il n'est pas nécessaire de gérer des événements à partir de la boîte de dialogue de fichiers. L'appel de `setVisible(true)` se bloque jusqu'à ce que l'utilisateur sélectionne OK. Le fichier sélectionné est renvoyé sous forme de chaîne. Voir `getDirectory()`, `getFile()`



ScrollPane

Fournit un conteneur général ne pouvant pas être utilisé de façon autonome. Il fournit une vue sur une zone plus large et des barres de défilement pour manipuler cette vue.

```
Frame f = new Frame("ScrollPane");
Panel p = new Panel();
ScrollPane sp = new ScrollPane();
p.setLayout(new GridLayout(3, 4));
sp.add(p);
f.add(sp);
f.setSize(200, 200);
f.setVisible(true);
```



Le `ScrollPane` crée et gère les barres de défilement selon les besoins. Il contient un seul composant et on ne peut pas influencer sur le gestionnaire de disposition qu'il utilise. Au lieu de cela, on doit lui ajouter un `Panel`, configurer le gestionnaire de disposition de ce `Panel` et placer les composants à l'intérieur de ce dernier.

En général, on ne gère pas d'événements dans un `ScrollPane`, mais on le fait dans les composants qu'il contient.

Menus

Les menus diffèrent des autres composants par un aspect essentiel. En général, on ne peut pas ajouter de menus à des conteneurs ordinaires et laisser le gestionnaire de disposition les gérer. On peut seulement ajouter des menus à des éléments spécifiques appelés conteneurs de menus. Généralement, on ne peut démarrer une "arborescence de menu" qu'en plaçant une barre de menus dans un Frame via la méthode `setMenuBar()`. A partir de là, on peut ajouter des menus à la barre de menus et incorporer des menus ou éléments de menu à ces menus.

L'exception est le menu `PopupMenu` qui peut être ajouté à n'importe quel composant, mais dans ce cas précis, il n'est pas question de disposition à proprement parler.

Menu Aide

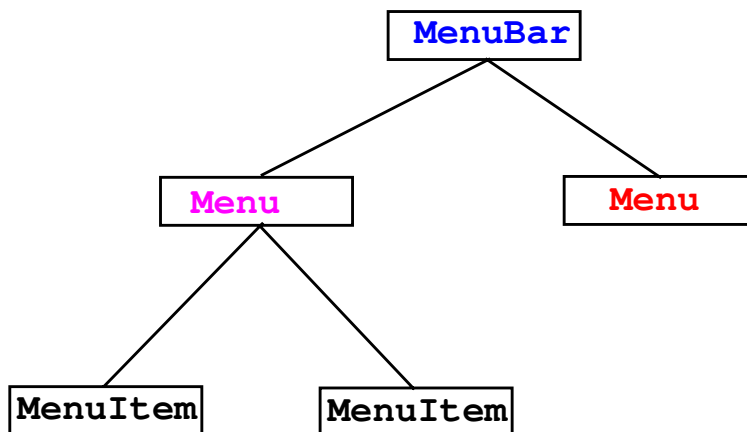
Une caractéristique particulière de la barre de menus est que l'on peut désigner un menu comme le menu Aide. Cette opération s'effectue par l'intermédiaire de la méthode `setHelpMenu(Menu)`. Le menu à considérer comme le menu Aide doit avoir été ajouté à la barre de menus, et il sera ensuite traité de la façon appropriée pour un menu Aide sur la plateforme locale. Pour les systèmes de type X/Motif, cela consiste à décaler l'entrée de menu à l'extrémité droite de la barre de menus.



MenuBar

C'est la barre de menu horizontale. Elle peut seulement être ajoutée à l'objet `Frame` et constitue la racine de toutes les arborescences de menus.

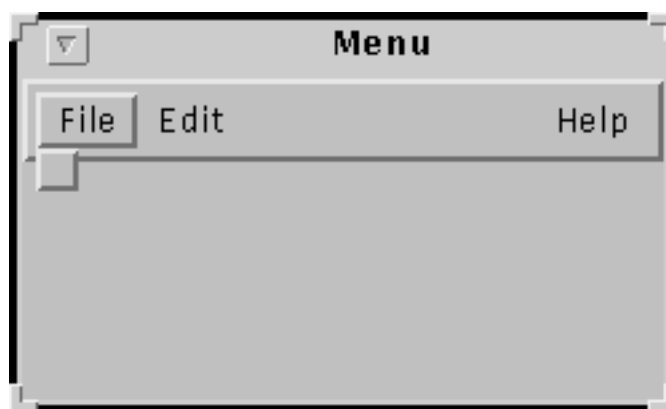
```
Frame f = new Frame("MenuBar");  
MenuBar mb = new MenuBar();  
f.setMenuBar(mb);
```



Menu

La classe `Menu` fournit le menu déroulant de base. Elle peut être ajoutée à une barre de menus ou à un autre menu.

```
MenuBar mb = new MenuBar();
Menu m1 = new Menu("File");
Menu m2 = new Menu("Edit");
Menu m3 = new Menu("Help");
mb.add(m1);
mb.add(m2);
mb.add(m3);
mb.setHelpMenu(m3);
```



Les menus présentés ici sont vides ce qui explique l'aspect du menu File.

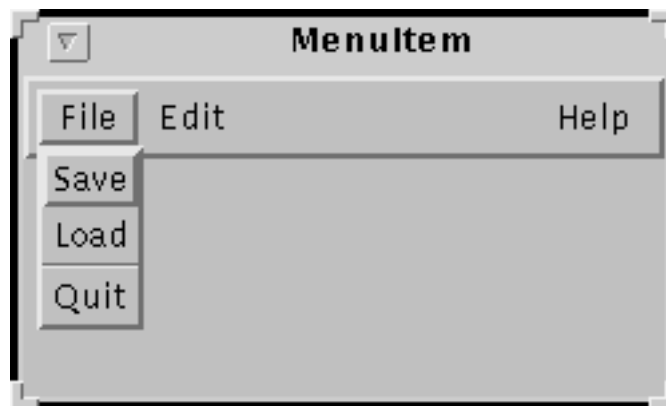
On peut ajouter un `ActionListener` à un objet `Menu`, mais c'est assez inhabituel. Normalement, les menus servent seulement à disposer des `MenuItem` décrits plus loin.



MenuItem

Les éléments de menu `MenuItem` sont les "feuilles" d'une arborescence de menu.

```
Menu m1 = new Menu("File");
MenuItem mi1 = new MenuItem("Save");
MenuItem mi2 = new MenuItem("Load");
MenuItem mi3 = new MenuItem("Quit");
m1.add(mi1);
m1.add(mi2);
m1.addSeparator();
m1.add(mi3);
```

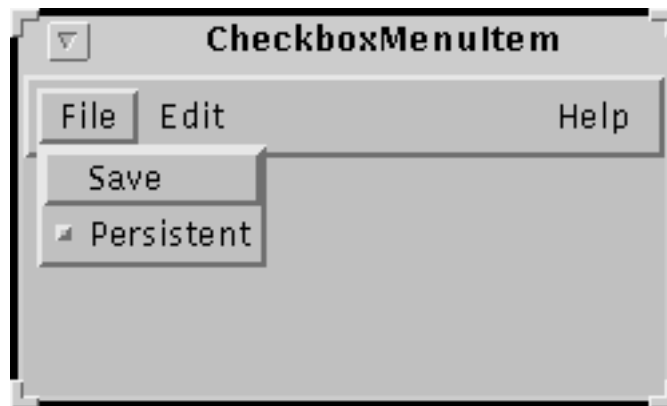


En règle générale, on ajoute un `ActionListener` aux objets `MenuItem` afin d'associer des comportements aux menus.

CheckboxMenuItem

Les éléments de menu à cocher permettent de proposer des sélections (activé/désactivé) dans les menus.

```
Menu m1 = new Menu("File");
MenuItem mi1 = new MenuItem("Save");
CheckboxMenuItem mi2 =
    new CheckboxMenuItem("Persistent");
m1.add(mi1);
m1.add(mi2);
```



L'élément de menu à cocher doit être surveillé via l'interface `ItemListener`. C'est pourquoi la méthode `itemStateChanged()` est appelée lorsque l'état de l'élément à cocher est modifié.



PopupMenu

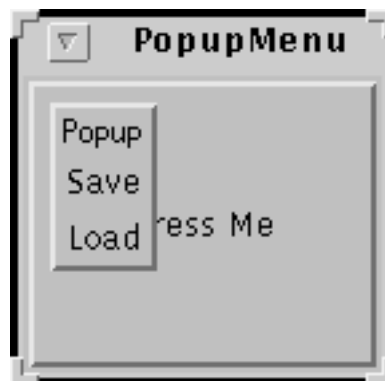
Fournit un menu autonome pouvant s'afficher instantanément sur un autre composant. On peut ajouter des menus ou éléments de menu à un menu instantané.

```
Frame f = new Frame("PopupMenu");
Button b = new Button("Press Me");
b.addActionListener(...);

PopupMenu p = new PopupMenu("PopupMenu");
MenuItem s = new MenuItem("Save");
MenuItem l = new MenuItem("Load");

s.addActionListener(...);
l.addActionListener(...);

f.add("Center", b);
p.add(s);
p.add(l);
f.add(p);
```



PopupMenu (suite)




Le menu PopUp doit être ajouté à un composant "parent". Cette opération diffère de l'ajout de composants ordinaires à des conteneurs. Dans l'exemple suivant, le menu instantané a été ajouté au Frame englobant.

Pour provoquer l'affichage du menu instantané, on doit appeler la méthode show. L'affichage nécessite qu'une référence à un composant joue le rôle d'origine pour les coordonnées x et y.

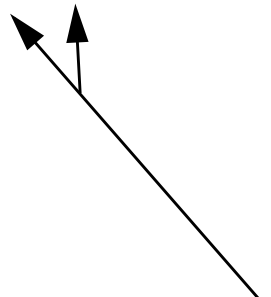
Dans cet exemple c'est le composant b qui sert de référence.

```
public void actionPerformed(ActionEvent ev) {  
    p.show(b, 10, 10);  
}
```

Composant de référence pour
l'affichage du popup menu



Coordonnées % composantes
d'origines



Le composant d'origine doit être sous (ou contenu dans) le composant parent dans la hiérarchie des composants.



Contrôle des aspects visuels

On peut contrôler l'apparence des composants AWT en matière de couleur de fond et de premier plan ainsi que de police utilisée pour le texte.

Couleurs

Deux méthodes permettent de définir les couleurs d'un composant :

- `setForeground(...)`
- `setBackground(...)`

Ces deux méthodes utilisent un argument qui est une instance de la classe `java.awt.Color`. On peut utiliser des couleurs de constante désignées par `Color.red` `Color.blue` etc. La gamme complète de couleurs prédéfinies est documentée dans la page relative à la classe `Color`.

Qui plus est, on peut créer une couleur spécifique de la façon suivante :

```
int r = 255, g = 255, b = 0;  
Color c = new Color(r, g, b);
```

Un tel constructeur crée une couleur d'après les intensités de rouge, vert et bleu spécifiées sur une échelle allant de 0 à 255 pour chacune.

Contrôle des aspects visuels

Polices

La police utilisée pour afficher du texte dans un composant peut être définie à l'aide de la méthode `setFont()`. L'argument utilisé pour cette méthode doit être une instance de la classe `java.awt.Font`.

Aucune constante n'est définie pour les polices, mais on peut créer une police en indiquant son nom, son style et sa taille en points.

```
Font f = new Font("TimesRoman", Font.PLAIN, 14);
```

Si la portabilité est recherchée il vaut mieux utiliser des noms de polices abstraites :

- Dialog, DialogInput
- SansSerif (remplace Helvetica)
- Serif (remplace TimesRoman)
- Monospaced (remplace Courier)
- Symbol

On peut obtenir la liste complète des polices en appelant la méthode `getFontList()` de la classe `Toolkit`. La boîte à outils (`toolkit`) peut être obtenue à partir du composant, une fois ce dernier affiché on appelle la méthode `getToolkit()`. On peut aussi utiliser le `Toolkit` par défaut obtenu par `Toolkit.getDefaultToolkit()`.



ATTENTION!: la méthode `getFontList` de `Toolkit` est obsolete en Java2; voir plutôt: `GraphicsEnvironment.getAvailableFontFamilyNames()`.



Contrôle des aspects visuels

Polices

Les constantes de style de police sont en réalité des valeurs entières (int), parmi celles citées ci-après :

- `Font.BOLD`
- `Font.ITALIC`
- `Font.PLAIN`
- `Font.BOLD + Font.ITALIC`

Les tailles en points doivent être définies avec une valeur entière.

Impression

L'impression est gérée dans Java 1.1 d'une façon similaire à l'affichage sur écran. Grace à une instance particulière de `java.awt.Graphics` toute instruction de dessin dans ce contexte est en fait destinée à l'imprimante.

Le système d'impression Java 1.1 permet d'utiliser les conventions d'impression locales, de sorte que l'utilisateur voit s'afficher une boîte de dialogue de sélection d'imprimante lorsque l'on lance une opération d'impression. L'utilisateur peut ensuite choisir dans cette boîte de dialogue les options telles que la taille du papier, la qualité d'impression et l'imprimante à utiliser.

```
Frame f = new Frame("Print test");
Toolkit t = f.getToolkit();
PrintJob job = t.getPrintJob(f, "Mon impr.", null);
if (job != null) {
    Graphics g = job.getGraphics();
    .....
}
```

Ces lignes créent un contexte graphique (`Graphics`) "connecté" à l'imprimante choisie par l'utilisateur. Pour obtenir un nouveau contexte pour chaque page :

```
f.printAll(g); // ou printComponents()
```

On peut utiliser n'importe quelle méthode de dessin de la classe `Graphics` pour écrire sur l'imprimante. Ou bien, comme indiqué ici, on peut simplement demander à un composant de se tracer. La méthode `print()` demande à un composant de se tracer, mais elle n'est liée qu'au composant pour lequel elle a été appelée. Dans le cas d'un conteneur, comme ici, on peut utiliser la méthode `printAll()` pour que le conteneur et tous les composants qu'il contient soient tracés sur l'imprimante. Utiliser `printComponents()` si on utilise des composants 100% JAVA.

```
g.dispose();
job.end();
```

Après avoir créé une page de sortie conforme à ce que l'on souhaite, on utilise la méthode `dispose()` pour que cette page soit soumise à l'imprimante.



Impression

Une fois ce travail terminé, on appelle la méthode `end()` sur l'objet tâche d'impression. Elle indique que la tâche d'impression est terminée et permet au système de traitement en différé d'exécuter réellement la tâche puis de libérer l'imprimante pour d'autres tâches.

Nota: Dans le contexte Solaris il n'existe pas d'objet standard de menu d'impression. On peut utiliser le 3ème argument de `getPrintJob()` qui est un dictionnaire de propriétés (qui peut aussi être `null`)

```
Frame f = new Frame("Print test");
Toolkit t = f.getToolkit();
Properties pprops = new Properties();
pprops.put("awt.print.paperSize", "a4");
pprops.put("awt.print.orientation", "landscape");
PrintJob job = t.getPrintJob(f, "Mon impr.", pprops);
.....
```

Les propriétés de ce dictionnaire sont :

```
awt.print.destination
awt.print.printer
awt.print.fileName
awt.print.options
awt.print.orientation (portrait, landscape)
awt.print.paperSize (letter, legal, executive, a4)
awt.print.numCopies
```



Le système d'impression change en Java 2 : voir package `java.awt.print`.



Points essentiels

Il existe de nombreux services associés à Java. Ces extensions sont souvent mises au point par des consortiums d'entreprises spécialisées dans un domaine d'application. Il faut faire la part de ce qui constitue des bibliothèques de composants, des API d'accès, des produits...

Quelques exemples de domaine:

- Graphique, Multimedia
- Réseau
- Utilitaires
- Accès données
- Echanges sécurisés
- Embarqué léger
- Système
- Produits divers
- exemple d'architecture : Java Enterprise



Graphique, Multimedia

Graphique

2 D : primitives sophistiquées pour dessin, texte, images avec gestion des couleurs, des opérations géométriques, de la composition. On peut utiliser texture, transparence, etc. Simplification de l'impression.

Support des niveaux de gris adaptés à l'imagerie médicale (voir également JAI).

3 D : programmation par description de scènes (graphe: objets géométriques, attributs, informations de visualisation, mouvements,...). Lié à un exécuteur optimisé.

Advanced Imaging API (JAI): chargement, traitement, composition et présentation d'images.

MultiMedia

MediaFramework : définit une architecture et une API pour le chargement et la présentation de media sous coordination temporelle (MPEG, MIDI, etc.). L'API "Media Player" supporte 3 niveaux : niveau client (contrôle simple de l'exécution), niveau personnalisé (le programmeur peut rajouter ou enrichir des fonctionnalités), niveau "design" (adjonction de nouveaux media et formats).

Sound : actuellement AudioClip de la plateforme Java2 fournit la possibilité d'exécuter différents formats audio (AIFF, AU, WAV) et MIDI (type0, type1, RMF), à l'avenir possibilité de synthèse de son.

Speech : API pour reconnaissance et synthèse de la parole. + Java Speech Grammar Format (JSGF) et Java Speech Markup Language (JSML).

Réseau

Dynamic management kit : Outil de développement d'agents liés à l'administration système/réseau. Permet de construire une administration hébergée par un navigateur WEB.

Contient des adaptateurs à de nombreux protocoles, un système de déploiement "push", des composants beans génériques, un compilateur de MIB (passerelles SNMP-Java), outil d'aide à la génération d'agents,...

Naming and directory interface : API d'accès à des services de nommage et d'annuaire. Permet de fédérer des services hétérogènes: LDAP, DNS, NIS, NIS+, NDS, RMI registry, COS Naming, etc.

Java Mail: API d'accès aux services du courrier électronique. Contient également une réalisation de référence pour l'accès à SMTP et IMAP.

Java Message service: API d'accès aux produits de communication par message (Message Oriented Middleware - IBM MQseries, TIBCO, etc.-).

Java Shared Data Toolkit: permet de développer des applications partageant interactivement des données sur un réseau. Exemples: outils de collaboration (tableau blanc partagé, conversation de groupe,...), visualisations "temps réel" (cours de bourse, ...), etc.

Java Spaces: partage de données, communications et coordination entre des objets Java situés sur un réseau. Paradigmes simples : write, read, take, notify,... . Facilités pour la persistance, pour des schémas divers de communication asynchrone (store & forward, filtres, etc.).

Jini: réseau applicatif dynamique. On "branche" un agent (logiciel, matériel) sur le réseau et il sera capable de : rechercher (et se joindre à) un groupe d'autres agents, publier ses capacités, rechercher des services, échanger des services dans un environnement sécurisé. Accès aux couches JavaSpaces et Transaction.



Utilitaires, composants

Java Communications API : protocole standardisé d'accès à des cartes de communication (port série RS232, parallèle IEEE1284).

Telephony : API établissement et gestion d'une communication téléphonique. (voir JavaPhone)

Infobus : sur une seule JVM échange de données entre des composants qui ne se connaissent pas *a priori*. Consommateurs et producteurs se reconnaissent par une étiquette commune et publient (ou recherchent) des données en indiquant la désignation et le type de codage (*flavor*) de ces données.

JavaBeans Activation Framework (JAF) : détermination d'un type de données chargées au runtime (depuis des ressources externes, depuis le réseau), accès transparent à ces données, découverte des opérations possibles sur ces données, activation des composants beans appropriés pour traiter ces données. Exemple : activation par un navigateur d'un bean approprié pour présenter/traiter un type de document reçu par http.

Enterprise JavaBeans: beans de niveau serveur (ou niveaux intermédiaires). Spécialisations des fonctions : sessions, communications, sécurité, recherche de données, transactions, syntonisation (*load balancing*, etc.) sont gérés par des composants différents de ceux qui gèrent la logique applicative associée au niveau courant.

Les EJB. persistents ou non (*session beans*, *entity beans*), sont administrés par des "Containers" qui fournissent (souvent de manière transparente) les services de base et activent, désactivent, sauvegardent les composants applicatifs.

"Don't rip and replace but wrap and embrace": politique recherche d'adaptateurs pour des applications existantes

Utilitaires programmation

JavaHelp : système de réalisation d'aide en ligne. Données et système de recherche peuvent résider coté client ou coté serveur.

HotJava HTML components (produit) : Composants beans permettant d'utiliser des fonctionnalités de navigateur WEB à l'intérieur d'une application.

Java project X (nom de code provisoire) : librairie d'accès à XML (analyseur, validation optionnelle, manipulation d'arbre en mémoire)

Java Server Engine (produit): composant serveur

JavaCC (Java compiler compiler) : permet de générer un analyseur à partir de la description d'une grammaire. La description comprend l'analyse lexicale et permet d'utiliser des grammaires sophistiquées.

APIs annexes de développement Java:

- **JBug** : accès programmatique au debug. Comprend : Java Debug Interface (JDI), Java Debug Wire protocol (JDWP), Virtual machine Debug Interface (JVMDI). JDI est l'interface de haut niveau 100%Java, JVMDI est l'interface de bas niveau fournie par la JVM.
- **Virtual Machine Profiler Interface** (JVMPI) : accès aux informations permettant de faire de l'analyse de performances.
- **Heap Analysis Tool** (HAT) : permet d'analyser des fichiers de *dump* générés à la demande par une machine virtuelle JAVA (-Xhprof) pour rechercher des références d'objets qui encombre la mémoire et ne sont par récupérés par le glaneur de mémoire (présence dans le code d'une chaîne intempestive de références fortes).

Ces Apis favorisent la réalisation d'ateliers de développement Java voir également paragraphe "produits divers"



Accès données

Java Transaction Service : Une API de bas niveau (selon la spécification CORBA *OTS*) pour l'accès aux services transactionnels. Destinée plutôt aux développeurs de serveurs ou de "Containers" d'EJB.

Java Transaction API : API de haut niveau pour la définition, au sein des applications, des "frontières" des transactions, des "Commit" à deux phases, etc... Contient un niveau d'accès aux services standard *Xa* de X/Open.

Java Blend : produit Sun/Baan. Outil de développement qui permet de générer des requêtes aux bases de données directement à partir des objets eux-mêmes et code exécutable qui optimise la gestion des requêtes au runtime.

StoreX (nom de code) : gestion des périphériques de stockage de données (*storage managment*).

Echanges sécurisés

SSL : Secure Socket Layer permet d'ouvrir des canaux de communication (socket voir `java.net`) sécurisés. `javax.net.ssl` est une API qui permet d'accéder à des produits (tierce partie) fournissant de tels services. La désignation d'URL "https" est directement comprise par la classe `java.net.URL`.

Java Cryptographic Extension (JCE): fournit un accès à des fonctions de cryptage/décryptage. Cet accès contient des aspects génériques et des aspects spécialisés en fonction des algorithmes choisis. La réalisation effective de ces services est fournie par Sun aux Etats-Unis/Canada et n'est pas exportable.

voir : <http://www.systemics.com>

Electronic Commerce Framework :

- **Java Wallet :**
 - Java Commerce Client : solution SUN de "Container" personnalisable pour des beans spécialisés. Charge des "Cassettes" (archives jar contenant des Commerce JavaBeans, des informations sur leur installation et des signatures identifiant des *Rôles*)
 - Java Commerce API: API de développement
 - Commerce JavaBeans :
 - Java Smart Card API : accès programmatique à des drivers permettant d'interroger des cartes du commerce (dont *javaring*)
- **Java Card API** (voir chapitre suivant)

Point Of Sale : évolutions de la "caisse enregistreuse".



Embarqué léger

PersonalJava : cibles : assistants personnels, téléphonie avec écrans (y compris mobiles haut de gamme), internet TV, consoles. Plage : ROM ~2M, RAM 512k-1M, processeurs : 32bits, >50mhz. Importance de la connection au réseau pour les services mais aussi pour la maintenance/mise à jour de la plate-forme.

Sous-ensemble de Java, compact, interfaces graphiques simplifiées (evt. sans souris ni clavier). Ecriture d'application liée à un savoir-faire : outils de contrôle -> PersonalJava Emulation Environment, JavaCheck.

API spécialisées: **JavaPhone, JavaTV, AutoJava**

EmbeddedJava : cibles : "pagers", instrumentation, téléphones bas de gamme,... Plage : ROM 256-512 K, RAM 256-512K, processeurs 16/32 bits > 25Mhz. Code Java moins généraliste et plus spécialisé.

Code disponible spécialisé et configurable. Adaptations de JVM : extensions "temps réel", gestion des ressources et de la mémoire.

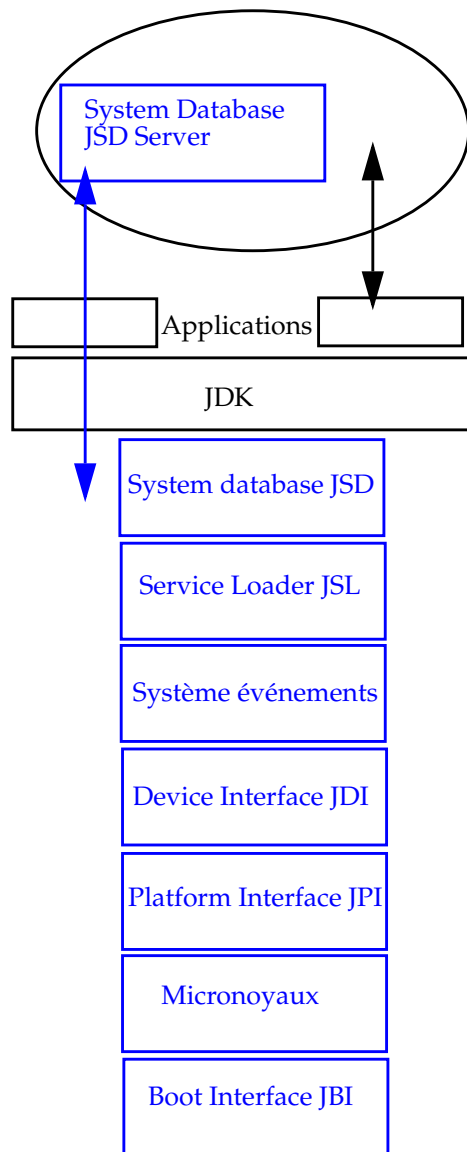
Nota: les regroupements de spécifications sont actuellement (fin 98) séparés en deux tendances une emmenée par Sun Microsystems et l'autre emmenée par HP.

JavaCard : cibles : cartes, Java Rings.

processeurs java : SUN développe un noyau de spécifications et de maquettage pour permettre aux licenciés de développer des processeurs spécialisés adaptés à des besoins particuliers.

Systeme

- **JavaOS** : petits OS tirant directement partie du matériel
- **JavaOS for Business** (Sun + IBM)
Os pour "client léger" avec administration centralisée. Le serveur gère les composants logiciels des applications ET du système.



Partie serveur : 100% Java, permet l'administration distante du système et des applications. Chargement et reconfiguration dynamique (événements).

Partie client : 70% Java, très modulaire techniques de modularité accessibles aux codes applicatifs support de JavaPOS

- **JavaOS for Consumers** : OS pour les machines de la catégorie PersonalJava



- JavaOs for NC :

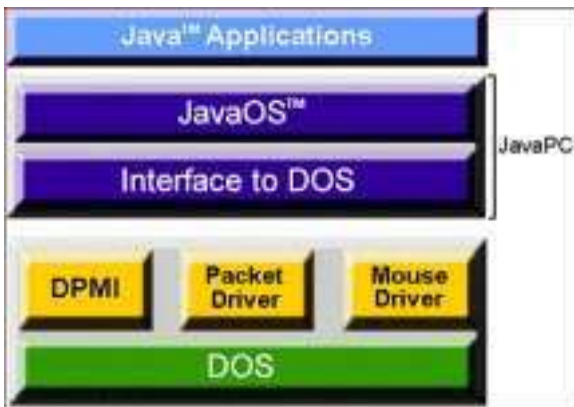


Le système est complètement téléchargé au démarrage.

L'interaction utilisateur se fait dans un cadre Hotjava/HotJavaViews

- adaptations "temps réel" : ChorusOS (Composants Système modulaires)

- JavaPC



Comment transformer un "vieux" PC en plateforme JavaOS

Produits divers

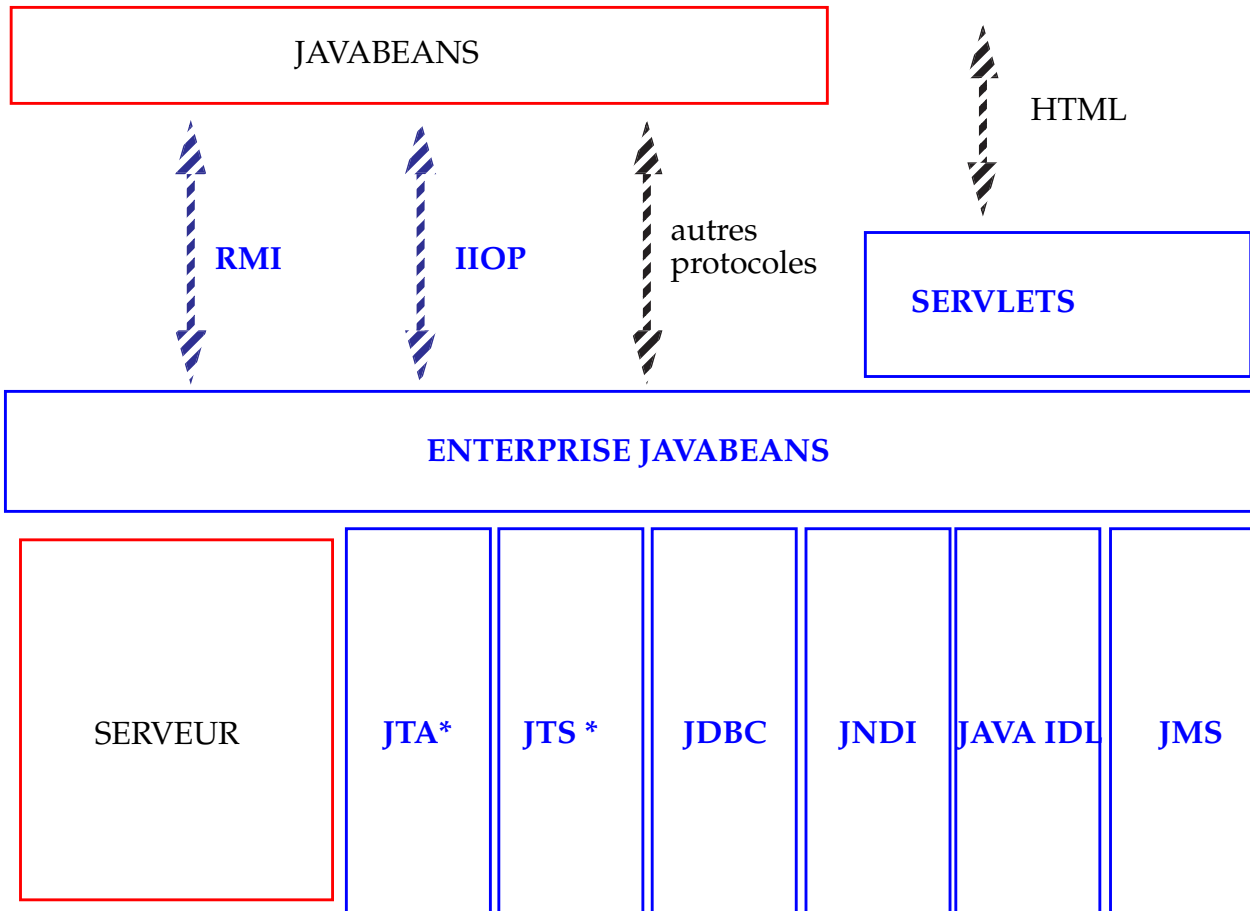
- **Outils de développement :**
Les Apis d'accès à la JVM facilitent le développement de nombreux outils intégrés. La capacité de Java à faire de l'intégration dynamique permet de créer des combinaisons d'outils. Une nouvelle catégorie d'outil est amenée à prendre une place stratégique : les "beans builder".
- **Outils de tests :** nombreux outils complémentaires de la filière de développement : analyse de charge, gestion de version, tests de non regression, etc. (voir par ex. produits SunTest)
- **Deploiement :**
 - **Java Plug-in :** permet de remplacer la JVM d'un navigateur par la dernière version de JRE.
"Plug-in HTML Converter" : Eventuellement transformation d'une applet pour téléchargement de cette nouvelle JVM.
 - Outils de *push* : ces produits (Marimba Castanet, PointCast,...) permettent une autre stratégie de déploiement. Une application (par ex. application client) est localement résidente; avant de démarrer elle vérifie avec le serveur si elle est conforme et, éventuellement, une mise à jour différentielle fine est effectuée.
- **Serveurs :**
 - **Java Embedded Server:** petit serveur permettant à des partenaires sur un réseau d'héberger des services (par ex. dispositifs dans la gamme PersonalJava)
 - **Serveurs d'applications :** fournissent un cadre intégré (administration, gestion des échanges, sécurité, sessions, accès données, transactions, etc.) à des composants applicatifs.





Java Enterprise APIs

Exemple d'architecture : **Java Enterprise APIs** regroupe un ensemble cohérent d'API pour la réalisation de serveurs d'entreprise :





La couverture des agences de Sun France permet de répondre à l'ensemble des besoins de nos clients sur le territoire.

Table 15.1 Liste des agences Sun Microsystems en france

Sun Microsystems France S.A 13, avenue Morane Saulnier BP 53 78142 VELIZY Cedex Tél : 01.30.67.50.00 Fax : 01.30.67.53.00	Agence d'Aix-en-Provence Parc Club du Golf Avenue G. de La Lauzière Zone Industrielle - Bât 22 13856 AIX-EN-PROVENCE Tél : 04.42.97.77.77 Fax : 04.42.39.71.52
Agence de Issy les Moulineaux Le Lombard 143, avenue de Verdun 92442 ISSY-LES-MOULINEAUX Ce- dex Tél : 01.41.33.17.00 Fax : 01.41.33.17.20	Agence de Lyon Immeuble Lips 151, boulevard de Stalingrad 69100 VILLEURBANNE Tél : 04.72.43.53.53 Fax : 04.72.43.53.40
Agence de Lille Tour Crédit Lyonnais 140 Boulevard de Turin 59777 EURALILLE Tél : 03.20.74.79.79 Fax : 03.20.74.79.80	Agence de Toulouse Immeuble Les Triades Bâtiment C - B.P. 456 31315 LABEGE Cedex Tél : 05.61.39.80.05 Fax : 05.61.39.83.43
Agence de Rennes Immeuble Atalis Z.A. du Vieux Pont 1, rue de Paris 35510 CESSON-SEVIGNE Tél : 02.99.83.46.46 Fax : 02.99.83.42.22	Agence de Strasbourg Parc des Tanneries 1, allée des Rossignols Bâtiment F - B.P. 20 67831 TANNERIES Cedex Tél : 03.88.10.47.00 Fax : 03.88.76.53.63
Bureau de Grenoble 32, chemin du Vieux Chêne 38240 MEYLAN Tél : 04.76.41.42.43 Fax : 04.76.41.42.41	