Maîtrise de la programmation en langage Java

Intitulé Cours : Maîtrise de la programmation en langage Java

Date : 4/6/02


Protections Juridiques

AVERTISSEMENT

Introduction

Objectifs

Ce cours a pour objectif de permettre à des programmeurs Java d'approfondir et d'élargir leur pratique des aspects fondamentaux du langage.

Les aspects conceptuels et techniques y sont abordés en parallèle. L'objectif est d'amener les professionnels vers une pratique mature de la programmation Java.

Sortent du périmètre de ce cours :

-L´étude détaillée de librairies très spécialisées comme sound, CORBA etc..

- Les approches Java sortant du périmètre "Standard Edition" comme Java Entreprise Edition, Java Micro-Edition, etc.

- Les librairies d'extension

Ces points sont abordés dans d'autres cours spécialisés


Prérequis

Pour pouvoir tirer un maximum de bénéfices de ce cours il est vivement conseillé:

- D'avoir une bonne pratique des aspects fondamentaux de Java[1]:
  - classes et interfaces, membres, constructeurs
  - héritage, polymorphisme
  - exceptions

- D'avoir de bonnes connaissances théoriques sur la programmation en général et la programmation objet en particulier.
- D'avoir des notions sur les technologies du Web internet (usage d'un navigateur, adresses URL, langage HTML)
- De connaître un minimum d'anglais (il faut pouvoir lire la documentation)

1. Ces notions sont développées dans notre cours SL-265 (*Programmation en langage Java*) et les cours dérivés SL-110 (*Langage Java pour programmeurs débutants*) et SL-010 (*fondamentaux java pour programmeurs objet*)

Phases

Le cours est décomposé en plusieurs parties :

1. Un bloc de chapitres revient sur les aspects fondamentaux de Java Les interfaces graphiques Swing servent de support à ces rappels qui sont complétés par un cours sur les processus légers (*thread*s).

2. Un bloc de chapitres introduit les relations avec l'environnement local d'exécution: archives, ressources, internationalisation, préférences, rapports et journalisation.

3. Un bloc de chapitres traite des systèmes d'entrées/sorties et d'échanges: différents systèmes d'E/S , programmation réseau, références distantes (RMI), accès aux bases relationelle(JDBC).

Chaque chapitre dispose d'annexes qui ne font pas à proprement parler partie du cours, mais qui permettent de signaler des informations complémentaires. Le support dispose également d'annexes globales qui abordent des dispositifs avancés de programmation comme la programmation dynamique, les références faibles ou les codes natifs.


Table des Matières

Première partie: aspects fondamentaux du langage

.


Introduction technique, composants Swing 1

Points essentiels

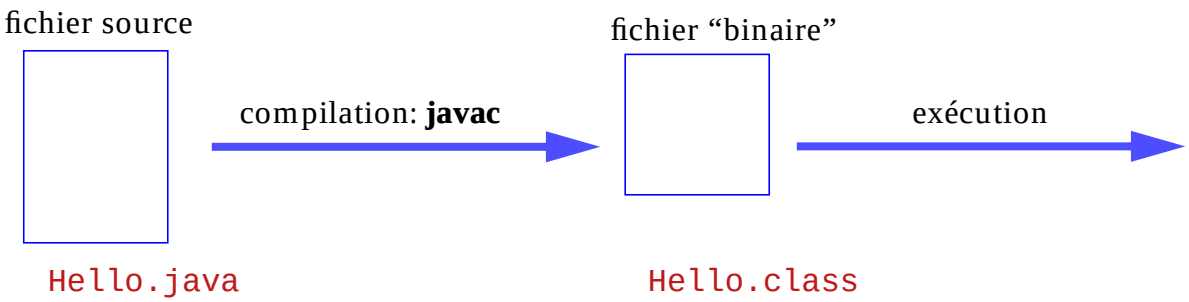
Première revue des bases techniques de Java.

Si nécessaire un court prologue peut être inséré ici: Voir *Java et le Web: les applets*, page 297.

Une présentation des composants graphiques Swing sert de trame: composants portables, philosophie d'une disposition de composants indépendante de la plate-forme.

Le code portable Java

La filière de production du code Java :

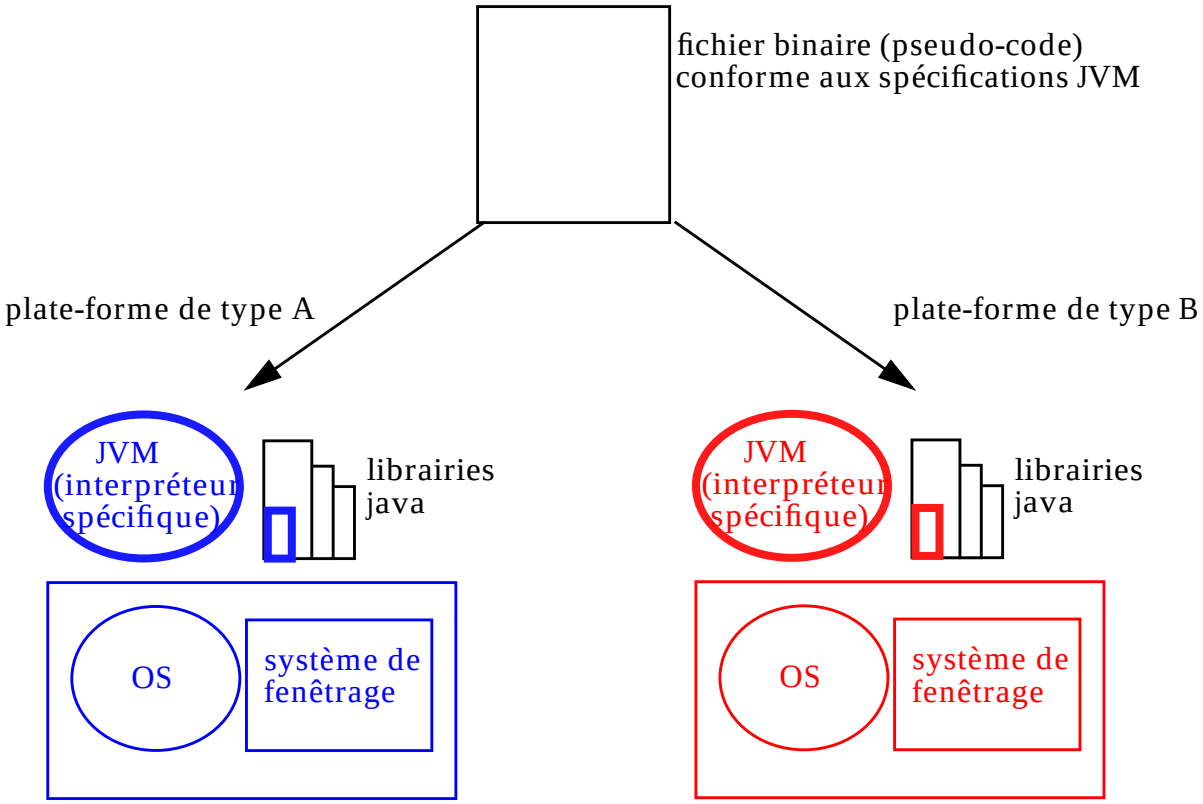


On a donc ici deux phases :

- Compilation du code source : `javac Hello.java`
- Exécution du pseudo-code généré: `java Hello` (appel de l’exécuteur avec en paramètre le nom de classe qui sert de point d’entrée) - on notera qu’il existe d’autres manières d’appeler l’exécuteur de code Java comme par exemple un exécuteur “enchassé” dans le code d’un navigateur. Dans ce cas on télécharge un code distant qui est dynamiquement pris en compte par l’exécuteur, le code doit répondre à un certain nombre de contraintes (dans ce cas celles décrites par la classe `java.applet.Applet`).

Les machines virtuelles

Le pseudo-code Java (*bytecode*) obéit à des spécifications précises. Portable, il est exécuté sur la plate-forme locale par un exécuteur propre à cette plate-forme



L'exécuteur local est un code "natif" spécifique au système local.

La librairie "système" de Java doit être installée localement. Cette librairie contient :

- des classes complètement écrites en Java
- des classes contenant du code "natif". Ces codes natifs permettent en particulier de déléguer des services au système d'exploitation local (ouverture de fichiers, etc.) ou au système de fenêtrage local (création d'une fenêtre, etc.)

Portabilité: exemple des applications graphiques

AWT

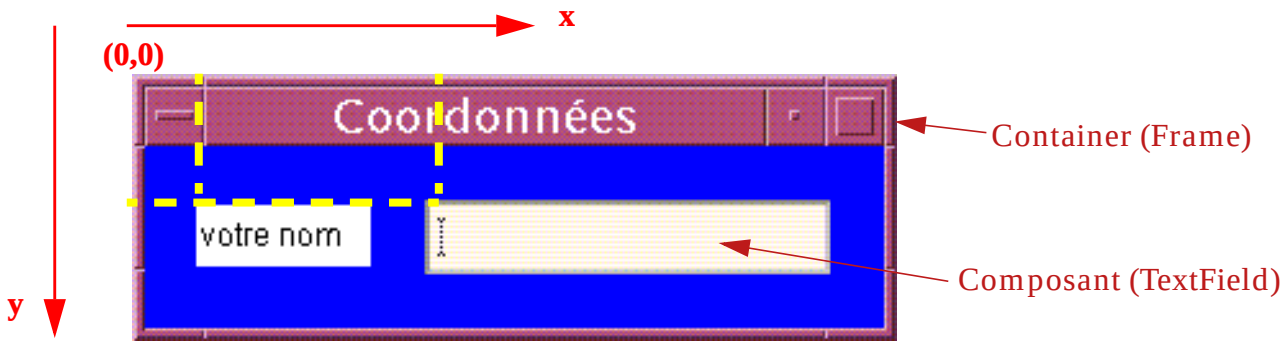
Le “package” `java.awt` définit un ensemble de composants graphiques standard que l’on retrouve sur toutes les plateformes. Ces composant d’interactions graphiques sont essentiellement réalisés avec des composants natifs du système de fenêtrage local. Le même code Java connaitra sur des sytèmes différents une réalisation sensiblement différente (tout en conservant la même logique):

Motif

Windows

Ici on a disposé deux boutons (classe `java.awt.Button`) “à l’intérieur” d’une fenêtre (classe `java.awt.Frame`).

AWT ne se contente pas de définir un vocabulaire graphique, il définit également une philosophie d’assemblage de ses composants. Dans la mesure où un code Java portable ne connait pas *a priori* la plateforme cible, il est malvenu de réaliser un positionnement absolu des composants (`Component`) à l’intérieur des `Container`s (composants qui en “contiennent” d’autre).



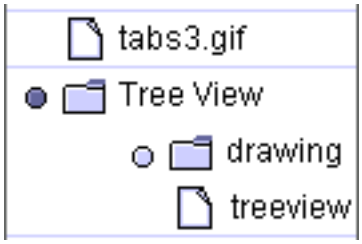
Bien que possible le positionnement en coordonnées absolues n’est pas portable

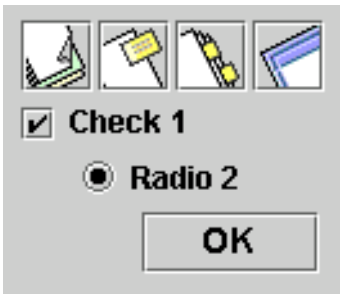
La disposition des composants s’appuie sur le principe des gestionnaires de disposition (`LayoutManager`): un objet associé à chaque container calcule la position des composants en fonction de contraintes logiques (nous allons en voir des exemples ultérieurement).

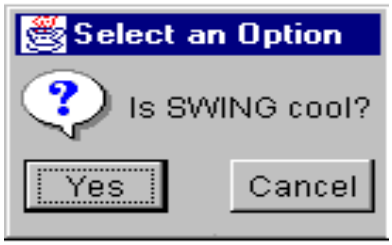
Swing

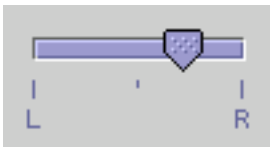
Le package `javax.swing` définit une autre catégorie de composants. Ces composants ne s'appuient pas sur des composants natifs mais sont "dessinés" à partir des primitives graphiques de bas niveau (accessibles en AWT).

On a ici un vocabulaire graphique plus riche (de par sa nature AWT a un vocabulaire limité), et de nombreuses techniques qui complètent celles d'AWT. Ainsi l'utilisation de gestionnaires de disposition est reprise et enrichie.









Pour une démonstration des capacités de la librairie voir le sous-répertoire `demo/jfc/SwingSet2` de l'installation de votre kit de développement java (SDK). On remarquera, entre autres, la capacité de changer de "*look and feel*", c'est à dire de passer d'une charte graphique portable (appelée `Metal`) au "*look and feel*" de la plateforme locale (voire à celui d'une autre plateforme comme un "look" Motif sur Windows par exemple).

Un premier programme Swing

Ici sur un système de fenêtrage Motif, le résultat de l'exécution d'un programme montrant un message simple dans une fenêtre:

Le fichier source `Message.java`

```
import javax.swing.* ; //directive utilisation de librairie

public class Message {//déclaration classe
   JFrame fenêtre ;

   public Message (String mess) {
      fenêtre = new JFrame("Message") ;
      JLabel label = new JLabel(mess) ;
      fenêtre.getContentPane().add(label);
   }

   public void démarrer() {
      fenêtre.pack();
      fenêtre.show() ;
   }
}// fin classe
```

var.membre instance

constructeur

méthode d'instance

Un autre fichier source (`TestMessage.java`) pour assurer le lancement de l'application :

```
public class TestMessage {
   public static void main (String[] args) {//point d'entrée
      Message messager = new Message(args[0]) ;
      messager.démarrer() ;
   }
}
```

La compilation des 2 fichiers donne les fichiers de pseudo-code : `Messsage.class` et `TestMessage.class`. Le lancement de l'application se fait par la commande (ici en syntaxe Shell Unix) :

```
java TestMessage " Bien le bonjour de SWING! "
```

Un premier programme Swing: contenu

Aspects fonctionnels

Dans une fenêtre Swing (`JFrame`: container “racine” à partir duquel on peut opérer en Swing) on dispose une étiquette Swing (`JLabel`: on ne peut pas mélanger composants AWT et composants Swing).

Comme pour tous les containers “racine” cette disposition (application de la méthode `add`) ne s’applique pas directement sur le `JFrame` mais sur son “*Content Pane*” (tapisserie de fond) -dans ce cas il y a une petite différence entre Swing et AWT qui, lui, dispose directement dans le Container Frame-.

La méthode `pack()` de `JFrame` lui permet de tenter de prendre les dimensions strictement nécessaires pour contenir les composants (effet plus portable que l’emploi de `setSize(largeur, hauteur)`).

Code java

La classe `Message` (dans le fichier `Message.java`!) définit une variable d’instance (`fenêtre`) qui est initialisée dans le constructeur et utilisée dans la méthode d’instance `démarrer` (méthode d’instance sans paramètre ni resultat).

La classe `TestMessage` contient juste un point d’entrée standard dont la description doit être :

```
public static void main(String[] parms)
```

Le tableau de chaînes de caractères passé en argument du `main` correspond aux arguments de lancement du programme. La première chaîne (index 0) est récupérée pour être passée en argument au constructeur de `Message`. Cette invocation du constructeur rend une reférence vers un objet de type `Message`; la méthode `démarrer` est invoquée sur cette instance.

Point intermédiaire : mise en place d'un programme

Mini-exercice :

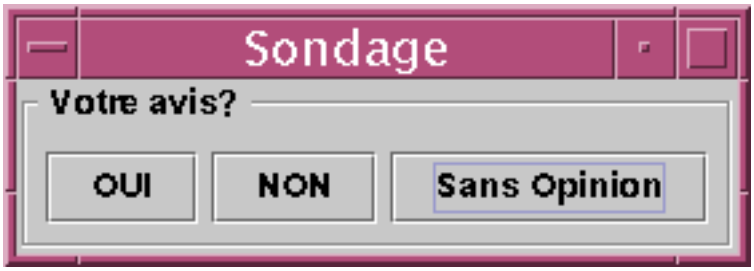
- En vous inspirant du modèle précédent réaliser un code Java Swing qui affiche un message. Compiler, exécuter.

  Pour varier les plaisirs utiliser pour afficher un message un autre composant que `JLabel` (par exemple un `JButton`, ou un `JTextField`) -lire la documentation associée au SDK-

- Pour les très rapides ou ceux qui révisent : changer la couleur de fond, la couleur du texte, la fonte du texte (méthodes `setBackground`, `setForeground`, `setFont` ,...)

Disposer des composants dans un Container

Voici un exemple plus complexe qui va nous permettre d'obtenir ceci:



Le code du Container dans lequel on va disposer des boutons :

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

public class Barre extends JPanel {

   /** Crée un barre horizontale de composants Swing
     * <P>
     * Les composant sont cadrés dans le sens naturel de lecture
     */
   public Barre(JComponent[] tbComp) {
      super(new FlowLayout(FlowLayout.LEADING)) ;
      for(int ix = 0 ; ix < tbComp.length; ix++) {
         this.add(tbComp[ix]) ;
      }
   }

   /**  Un cadre et un titre sont diposés autour des composants
     */
   public Barre(String leTitre, JComponent[] tbComp) {
      this(tbComp) ;
      Border bordure = BorderFactory.createEtchedBorder() ;
      Border titre = BorderFactory.createTitledBorder(
            bordure, leTitre,TitledBorder.DEFAULT_JUSTIFICATION,
            TitledBorder.TOP) ;
      this.setBorder(titre) ;
   }
}
```

Disposition : le code

Aspects fonctionnels

La classe `Barre` décrit un Panneau (`JPanel`) dans lequel les composants sont disposés les uns à la suite des autres.

Une telle disposition est assurée par un gestionnaire de disposition appelé **`FlowLayout`**. Pour doter un container d'un gestionaire de disposition particulier, il faut faire appel à sa méthode `setLayout(gestionnaire)`. La classe `JPanel` dispose également d'un constructeur qui permet de passer le gestionnaire de disposition en paramètre.

Dans le cadre d'un `FlowLayout` on doit "ajouter" les composants au container en employant la méthode `add(composant)` -l'ordre des ajouts est important puisqu'il détermine l'ordre de disposition des composants-

Il est possible de construire deux types de `Barre`s: celles qui alignent simplement des composants et celles qui entourent cet alignement avec une bordure comprenant un titre (cas illustré dans l'image). Cette possibilité de placer une bordure autour d'un composant est caractéristique de Swing (et n'existe pas en AWT classique).

Code java

`Barre` hérite de `JPanel` (déclaration `extends`). On décrit deux constructeurs dont l'un se définit par rapport à l'autre:

- Le premier constructeur fait appel au constructeur de `JPanel` en lui passant une instance de `FlowLayout` (argument déterminant un cadrage à gauche: constante de la classe FlowLayout).
  On parcourt ensuite le tableau de composants passé en argument et on les "ajoute" à l'instance courante (qui est un Container).

- Le second contructeur fait d'abord appel au premier, puis construit une bordure composée.
  Ces bordures sont fabriquées par des méthodes de classe de la classe `BorderFactory` (méthodes "fabriques").

Disposer des composants dans un Container (suite)

Le code utilisant `Barre` pour créer l'image en exemple:

```
import javax.swing.* ;

public class TestBarre {

   public static void main (String[] args) {
      JComponent[] tb = {
         new JButton("OUI"),
         new JButton("NON"),
         new JButton("Sans Opinion"),
      } ;
      Barre barre = new Barre(" Votre avis? ", tb) ;
      JFrame fen = new JFrame("Sondage") ;
      fen.getContentPane().add(barre) ;
      fen.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE) ;
      fen.pack();
      fen.show() ;
   }
}
```

Aspects fonctionnels

Un tableau contenant 3 boutons est passé au constructeur de `Barre` et cette Barre est mise directement dans le `ContentPane` de la fenêtre.

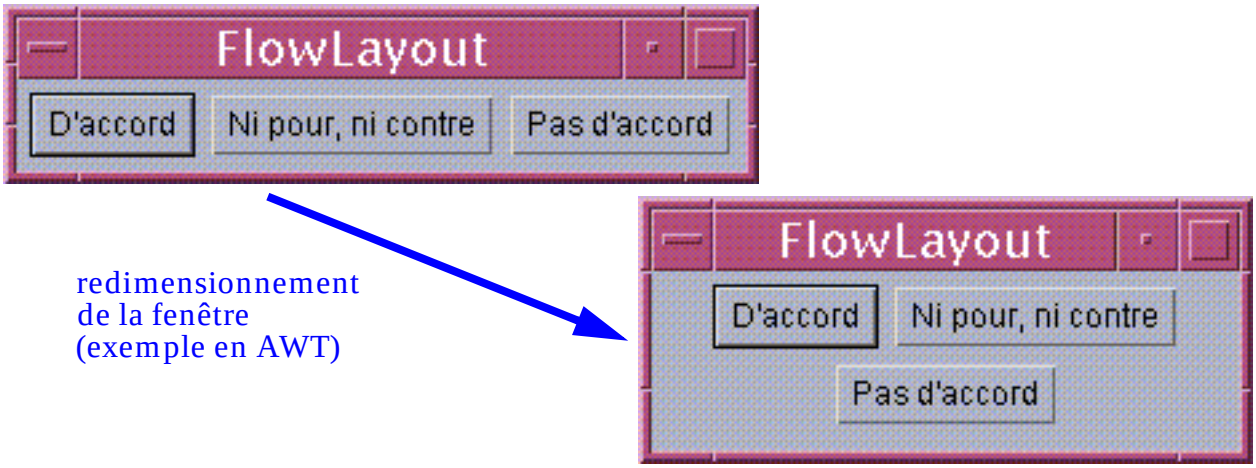
On notera que pour que l'application s'arrête quand on ferme la fenêtre, on fixe l'opération à déclencher (fonctionnalité spécifique au JFrame: dans le AWT classique il faudra suivre une procédure décrite au chapitre suivant).

Code java

Noter la création d'un tableau "littéral" de `JComponent` (qui d'ailleurs ne contient que des `JButtons`).

FlowLayout : synthèse

Le gestionnaire de disposition FlowLayout positionne les composants “les uns à la suite des autres”. Au besoin il crée une nouvelle ligne pour positionner les composants qui ne tiennent pas dans la ligne courante.



A la différence de beaucoup d’autres gestionnaires de disposition, une instance de FlowLayout ne modifie pas la taille des composants (tous les composants comportent une méthode `getPreferredSize()` qui est appelée par les gestionnaires de disposition pour demander quelle taille le composant voudrait avoir).

- `FlowLayout` est le gestionnaire par défaut associé aux containers de type Panel (AWT) et JPanel (Swing)[1].

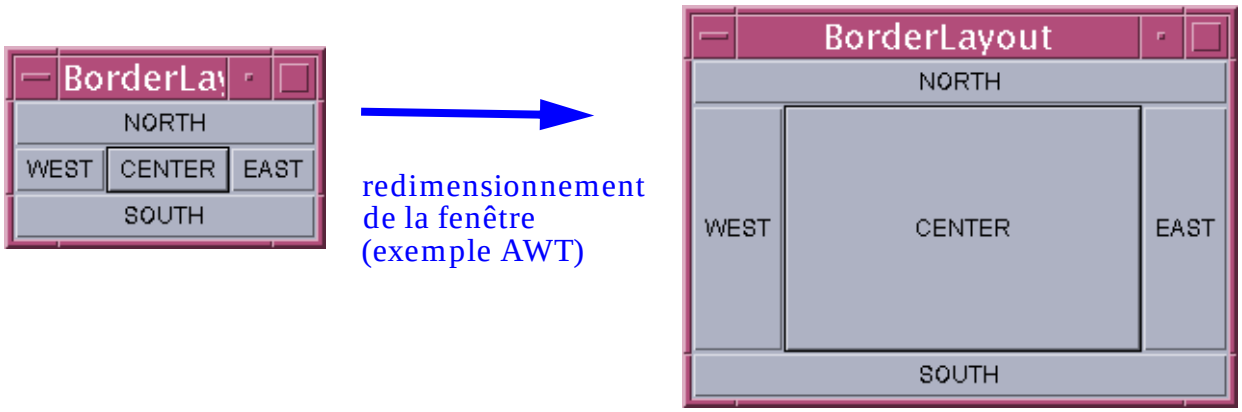
- Il existe plusieurs manières de construire un FLowLayout ce qui permet de faire varier l’alignement (à gauche, à droite, centré,..) ou de faire varier la taille des gouttières séparant les composants.

- Les composants sont ajoutés au Container en employant la méthode `add(Component)` , bien entendu l’ordre des `add` est important puisqu’il détermine l’ordre des composants dans la disposition.

1. note: un autre gestionnaire par défaut est associé au `ContentPane` le `add` n’a pas ici le même effet -voir suite de l’exposé-

Gestionnaires de disposition :BorderLayout

`BorderLayout` divise son espace de travail en cinq zone nommées et les composants sont ajoutés explicitement à une zone donnée. On ne peut placer qu'un composant par zone, mais toutes les zones ne sont pas nécessairement occupées.



- `BorderLayout` est le gestionnaire par défaut des Containers de type `Window` (AWT) et ContentPane des `JFrame` (Swing).

- Les `add` sont qualifiés: il faut préciser dans quelle zone on met le composant (par défaut un `add` simple met le composant dans la zone centrale).

- En cas d'élargissement du container le gestionnaire respecte les hauteurs "préférées" des composants en NORTH et SOUTH, et les largeurs "préférées" des composants en EAST et WEST. La zone centrale tend à occuper toute la place restante dans les deux directions.

BorderLayout: exemple de code (simplifié)

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

public class PresentationMessage extends JPanel {

  ... ;
   protected JButton rep= new JButton("Répondre") ;
   protected JButton prec= new JButton("Précédent") ;
   protected JButton suiv= new JButton("Suivant") ;

   public PresentationMessage() {
      ... ;
      setLayout(new BorderLayout(5,5)) ;// gouttières de 5
      // ---------- en-tête
      String descr = ... ;
      JLabel label = new JLabel(descr) ;
      this.add(label, BorderLayout.NORTH) ;
      // ---------- TEXTE
       String message = ... ;
      JTextArea jt = new JTextArea(message, 3, 50) ;
      this.add(new JScrollPane(jt), BorderLayout.CENTER) ;
      // ---------- boutons
      JButton[] tbBout = {rep, prec, suiv};
      Barre boutons = new Barre(tbBout) ;
      this.add(boutons, BorderLayout.SOUTH) ;
   }
....
}
```

On notera que le `add` prend ici une forme particulière (add “qualifié”).Par ailleurs la combinaison `JTextArea` (texte multiligne), `JScrollPane` est spécifique à Swing.

BorderLayout : exemple développé

Voici l'aspect graphique de la présentation recherchée :

Cette présentation graphique sert aux interactions sur un objet `Message` dont voici la définition:

```
package chat;// la class appartient au package "chat"

public class Message {// "vrai" nom : chat.Message
   public static final int ORIGINAL = -1 ;
   public final int numero ;
   public final int repondA ;
   public final String auteur ;
   public final String message ;

   public Message(String auteur,int num,int rep,String message){
      this.auteur=auteur; numero = num;
      repondA = rep; this.message = message;
   }

   public Message(String auteur,int num,String message) {
      this(auteur, num, ORIGINAL, message) ;
   }
}
```

Cette classe définit en quelque sorte une "structure de données" simple dont tous les champs sont immuables une fois l'instance construite. Du coup il n'y a pas encapsulation des variables d'instance et celles-ci sont publiques. (On notera également la définition de la constante de classe `ORIGINAL`).

Toute classe Java qui sort du domaine de la petite démonstration **doit** être définie dans le cadre d'un package(Voir *Packages, règles de responsabilité*, page 340..)

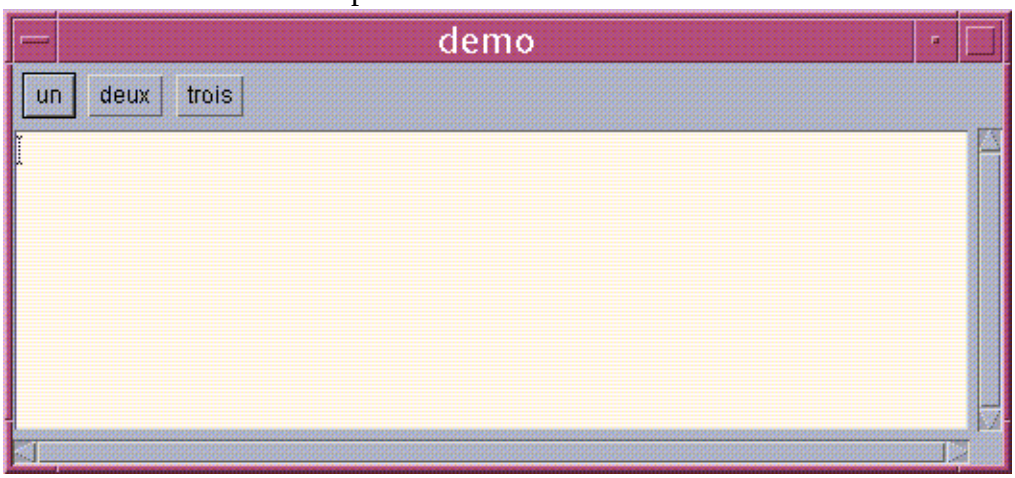
Exemple développé de l'interface graphique:

```
package chat;
import javax.swing.* ;
import javax.swing.text.* ;
import java.awt.* ;

public class PresentationMessage extends JPanel {

   private Message msg ;
   protected JButton rep= new JButton("Répondre") ;
   protected JButton prec= new JButton("Précédent") ;
   protected JButton suiv= new JButton("Suivant") ;

   public PresentationMessage(Message mess) {
      msg = mess ;
      setLayout(new BorderLayout(5,5)) ;// gouttières de 5
      // ---------- en-tête
      String descr =  "De: " +msg.auteur +"  ["+ msg.numero+ "] ";
      if( msg.repondA != Message.ORIGINAL) {
         descr += " répond A: [" + msg.repondA +"]" ;
      }
      JLabel label = new JLabel(descr) ;
      label.setFont(new Font("Monospaced",Font.BOLD,15)) ;
      label.setForeground( Color.RED ) ;
      this.add(label, BorderLayout.NORTH) ;
      // ---------- TEXTE
      JTextArea jt = new JTextArea(msg.message, 3, 50) ;
      jt.setEditable(false) ;
      this.add(new JScrollPane(jt), BorderLayout.CENTER) ;
      // ---------- boutons
      JButton[] tbBout = {rep, prec, suiv};
      Barre boutons = new Barre(tbBout) ;
      this.add(boutons, BorderLayout.SOUTH) ;
   }

   public Message getMessage() {// accesseur d'un membre privé
      return msg ;
   }
}
```

Point intermédiaire : coder avec des LayoutManagers

Mini-exercice :

- Réaliser un code Java Swing qui soit l'équivalent de cette présentation AWT:



Faites un code simple, mais les classes utilisées doivent appartenir à un package. Si vous ne savez pas organiser un développement en package: Voir *Développement avec des packages*, page 43..

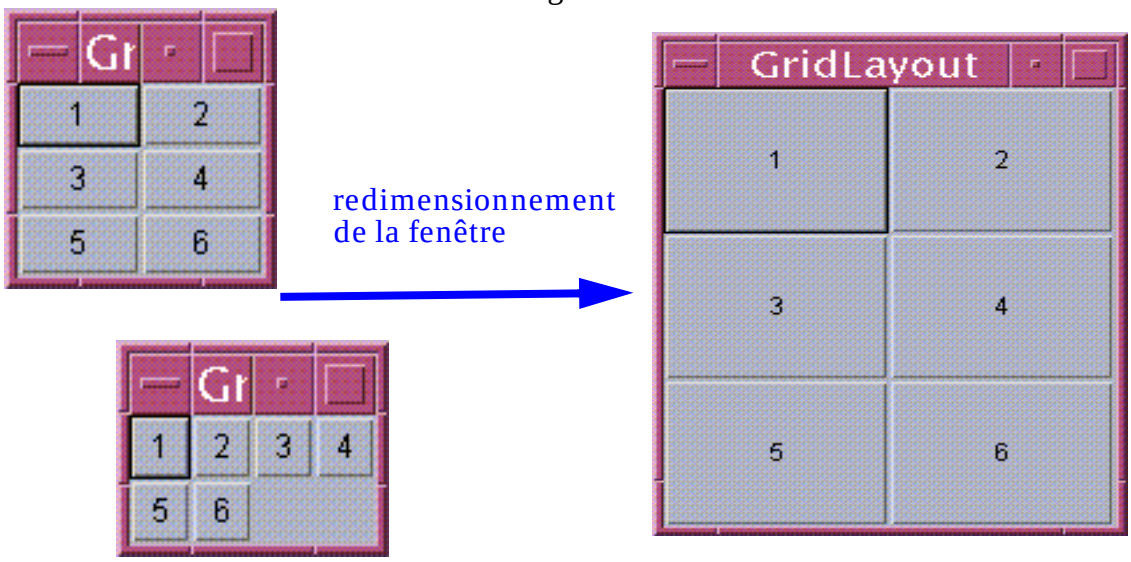
- Pour les très rapides ou ceux qui révisent : faites des expériences avec des composants Swing (JLabel, JButton) en leur passant du html comme texte.
  Ci-après un exemple analogue au code précédent mais qui utilise un `JEditorPane("text/html", texte)` en remplacement d'un TextArea.


Vous avez un Message!
De: Maistre Alcofribas Nasier [10] répond A: [3]
FAY CE QUE VOUDRAS
Cy n'entrez pas, hypocrites, bigotz,
Vieux matagotz, marmiteux borsouflés,
Torcoulx, badaux, plus que n'estoient les Gots,
Répondre
Précédent
Suivant

Quelques autres gestionnaires de disposition

GridLayout

Dispose un “tableau” dans lequel tous les composants ont la même hauteur et la même largeur:

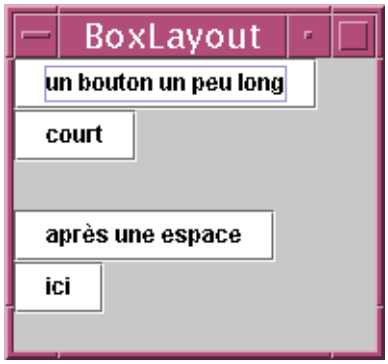


disposition avec des éléments manquants

Utilise essentiellement des `add(composant)` -l’ordre des appels étant important pour déterminer l’ordre de composants-

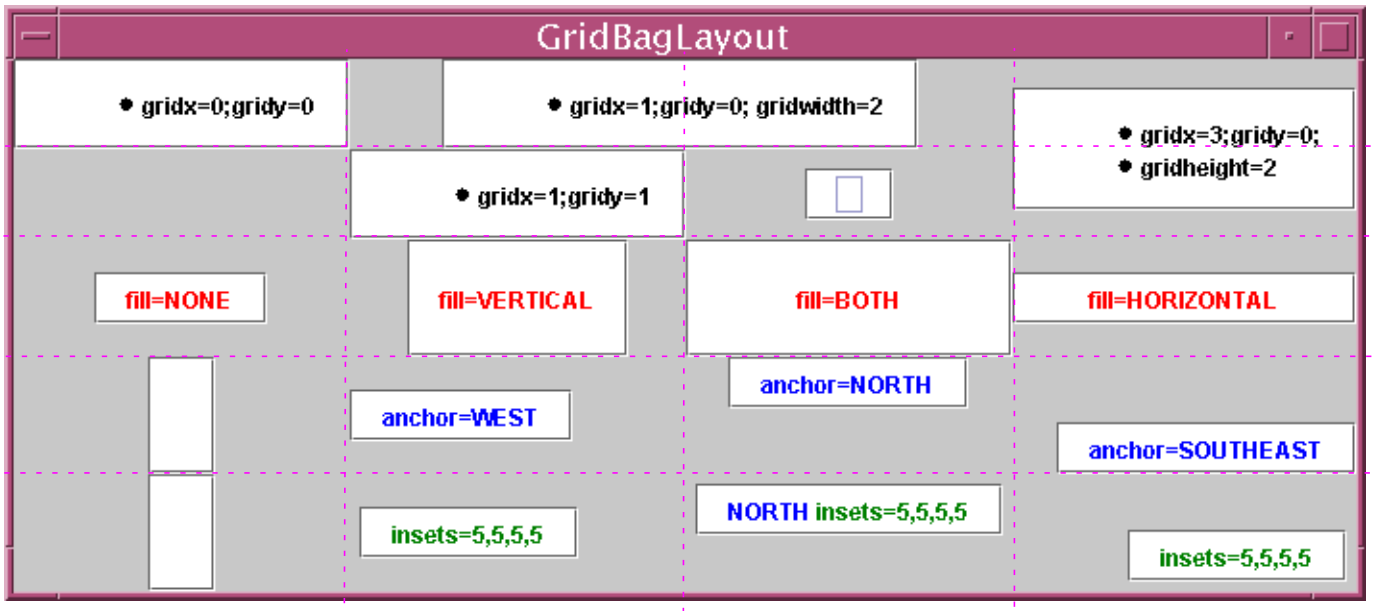
BoxLayout

C’est un gestionnaire spécifique à Swing. A utiliser éventuellement avec la classe Box pour permettre des alignements de composants dans une direction avec des insertions d’espaces et de “glue” qui maintient des éléments ensembles.



GridBagLayout

C’est le plus complexe des gestionnaires standard. Les composants sont disposés dans un tableau dans lequel chaque ligne et chaque colonne a une dimension dictée par le plus grand composant qui l’occupe (définissant ainsi un “tartan” irrégulier de cellules) :



On dispose par `add(composant, contraintes)` dans lequel *contraintes* est une instance de `GridBagConstraints` (réutilisable), les champs de cette instance comprennent:

- *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)
- *gridwidth, gridheight* : nombre de cellules occupées par le composant
- *fill* : direction du remplissage (le composant tend alors à occuper toute sa cellule dans la direction donnée). Valeurs: NONE, BOTH, VERTICAL, HORIZONTAL
- *anchor*: lorsqu'un composant est plus petit que sa cellule, bord d'ancrage du composant (un point cardinal: EAST, NORTHEAST, etc.)
- *insets*: détermination des "gouttières" (distance minimum entre le composant et les frontières de sa cellule)
- *weightx,weighty* : "poids" relatif de la cellule lorsque le container dispose de plus de place que nécessaire pour les cellules et que l'on souhaite élargir la disposition (valeur de type `double` comprise entre 0 et 1)

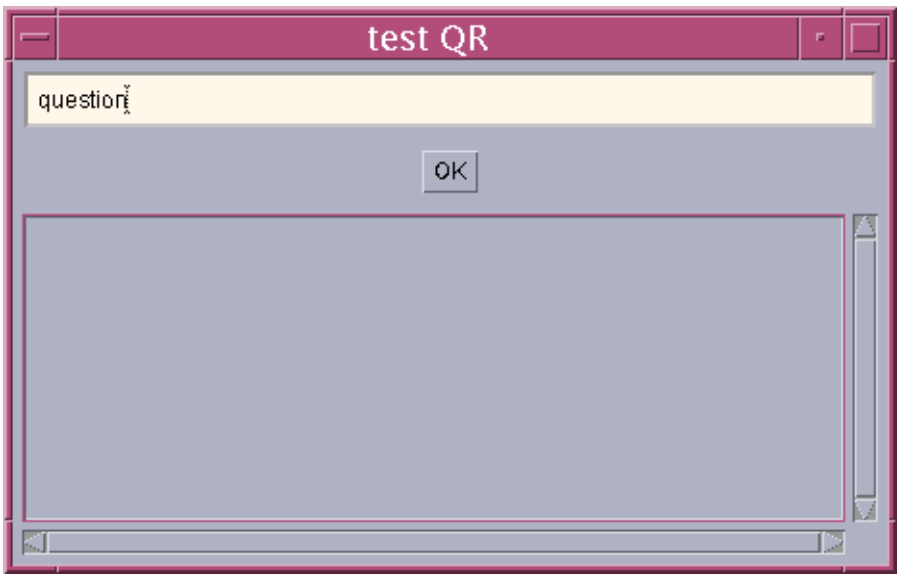
Exemple de code :

```
GridBagConstraints gbc = new GridBagConstraints() ;
gbc.gridx = 0 ; gbc.gridy = 0 ;
add(new JButton ("<html><li>gridx=0;gridy=0"), gbc);// couteux!
gbc.gridx = 1 ; gbc.gridy = 1 ;
add(new JButton ("<html><li>gridx=1;gridy=1"), gbc) ;
```

Exercice: pratique des LayoutManagers

Exercice :

- Réaliser en Swing un formulaire comprenant un champ de saisie, un bouton et une plage de texte (pour afficher la réponse à la question)



Compléments

Les compléments techniques suivants constituent une annexe de référence

Swing

La programmation Swing est un vaste sujet (voir notre cours SL-320). Quelques remarques générales complémentaires sur les composants Swing:

- **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 la charte graphique ("*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.

- **utilisation de modèles**:
  Swing comporte une très grande variété de composants dont certains sont très complexes (voir `JTree`, `JTable`, etc.). Il est pratique d'utiliser des "modèles" de programmation, c'est à dire des structures de données qui gèrent en parallèle des données et leur représentation graphique (les modifications de l'une sont répercutées dans l'autre).

Java: bases fondamentales

Pour améliorer vos connaissances sur les bases fondamentales de Java, vous devez, à ce stade, approfondir les points suivants :

- Architecture des déclarations de classes -Voir *Plan général d'une classe*, page 312.-, packages, règles de visibilité (modificateurs `public`, `private`, `protected`) -Voir *Packages, règles de responsabilité*, page 340.-.

- Utilisation des types: types "primitifs" (`int`, `boolean`, `double`, etc.), types "objets" (références vers des instances) et cas particulier des objets qui sont des tableaux. -Voir *Types primitifs scalaires, types objets*, page 339.

- Déclaration des variables --Voir *Déclarations de variable membre :*, page 314.; Voir *Variables*, page 322.-

- Déclaration des méthodes (arguments, résultats) -Voir *Déclarations de méthode membre*, page 315.; Voir *Méthodes*, page 329.-, règles de redéfinition dans les sous-classes. Polymorphisme des références.

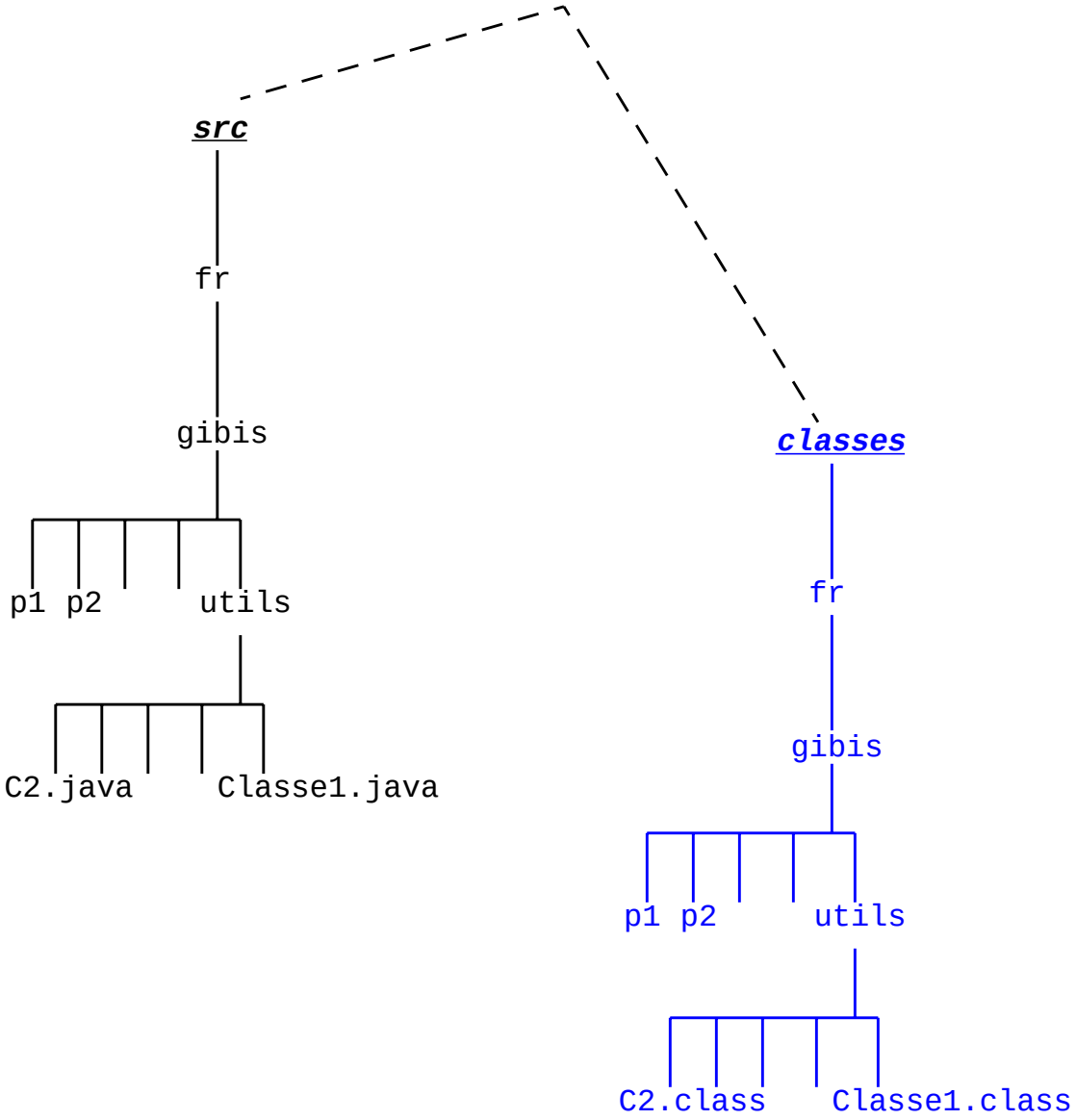
- Déclaration des constructeurs - Voir *Déclarations de constructeur*, page 317.; Voir *Constructeurs*, page 333. -, appel des constructeurs, règles de définition des constructeurs dans les sous-classes.

- Membres de classe : variables partagées -Voir *variables de classe*, page 325.-, constantes, méthodes de classe -Voir *Méthodes de classe*, page 332.-.

- Structures du code -Voir *Syntaxe simplifiée des blocs de code*, page 318.; Voir *Blocs*, page 335.-, structures de contrôle -Voir *Syntaxe simplifiée des blocs de code*, page 318.; Voir *structures de contrôle:*, page 319.

Développement avec des packages

Exemple d'organisation pratique des répertoires:




```
javac -classpath classes -d classes -sourcepath src Classe1.java
```

peut contenir une liste de chemins d'accès (séparés par ":" -ou ";" sous Win*-)

variables

Introduction technique (suite), événements 2

Points essentiels

La gestion des événements dans une IHM va nous permettre de revenir sur des dispositifs de programmation objet spécifiques à Java: interfaces, classes membres et classes locales.

Les mécanismes fondamentaux des événements graphiques relèvent de AWT, mais sont repris et élargis dans Swing.

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.

Sources d'événements

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

- `getActionCommand()` : renvoie le nom de commande associé à l'action.
- `getModifiers()` : renvoie la combinaison des "modificateurs", c'est à dire la combinaison des touches que l'on a maintenues pressées pendant le click (touche Shift par exemple).
- `getSource()` : rend l'objet qui a généré l'événement.

Gestionnaire d'événements

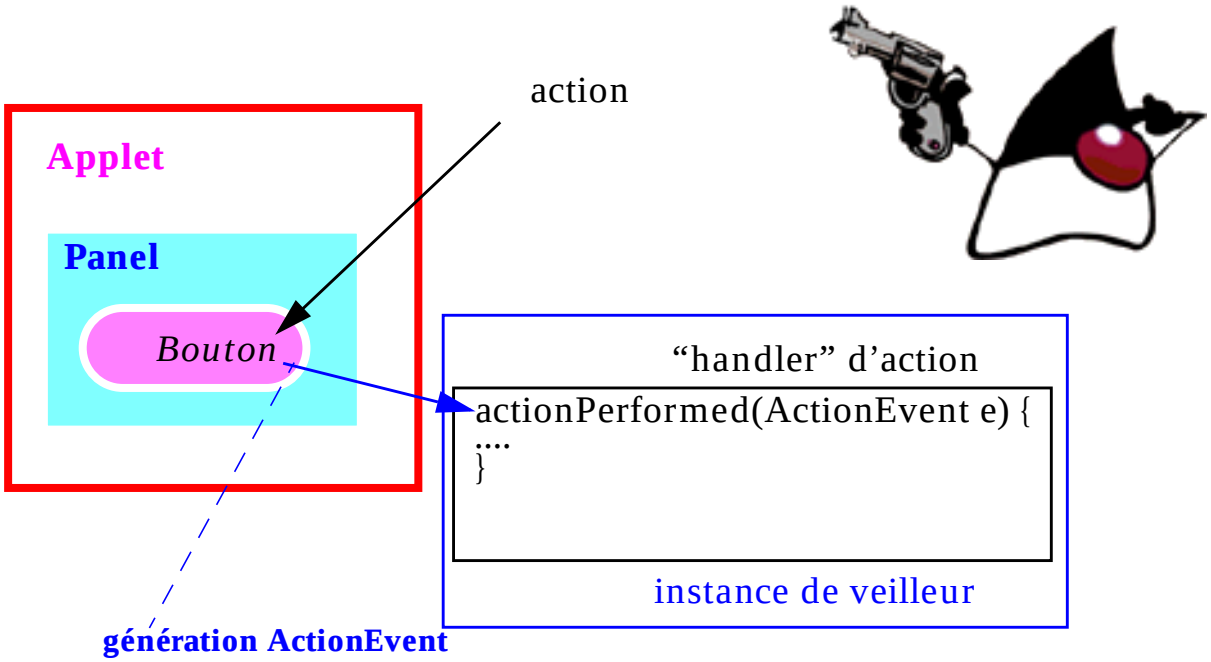
Un "gestionnaire" (*handler*) d'événement est une méthode qui reçoit un objet `Event`, l'analyse et traite les conséquences de l'interaction utilisateur.

Pour certains types d'événements il peut y avoir plusieurs types de gestionnaires qui seront respectivement appelés en fonction de l'action qui s'est produite au niveau du composant source. Ainsi pour un "événement souris" on pourra distinguer un gestionnaire correspondant à l'action du "clic" ou un gestionnaire correspondant au fait que le curseur soit passé sur le composant concerné.

Modèle d'événements

Modèle par délégation

Le JDK 1.1 a introduit dans Java un modèle d'événement appelé modèle d'événement par délégation. Dans un modèle d'événement par délégation, les événements sont (indirectement) générés par un composant graphique (qualifié de "*source*" de l'événement); on doit associer à ce composant un objet de traitement d'événement (appelé veilleur: `Listener`) pour recevoir l'événement et le traiter. De cette façon, le traitement d'événement peut figurer dans une classe distincte du composant impliqué.



Un "veilleur" est susceptible de traiter plusieurs actions liées à un type d'événement et donc de disposer de plusieurs gestionnaires : par exemple un gestionnaire chargé de veiller sur les "clics" souris et un chargé de veiller sur le passage de la souris sur le composant.

Modèle d'événements

Le type du "veilleur"

Les composants susceptibles de générer des événements sont dotés de méthodes d'enregistrement des veilleurs spécialisés. On peut avoir plusieurs méthodes d'enregistrement par exemple pour un bouton `JButton`:

```
addActionListener(...) ; addMouseListener(...) ;...
```

Quelle est le type de l'argument qui permet de désigner la nature du veilleur?

- Si ce type était celui d'une classe particulière on serait obligé au niveau de la réalisation de créer une sous-classe pour implanter le code de traitement.
  Bien que possible, cette solution serait plus rigide: on pourrait avoir besoin d'insérer ces codes de traitement dans des instances qui dérivent d'autres classes (et, pourquoi pas, des instances qui sont elles mêmes des dérivés de composants graphiques). De plus si on voulait créer une classe de veille qui a plusieurs comportement (ActionListener, MouseListener) l'écriture du code deviendrait difficile (pas d'héritage multiple en Java).
  On souhaiterait avoir ici à la fois un contrôle de type strict (l'objet passé en paramètre doit savoir traiter l'événement considéré) et une grande souplesse au niveau du type effectif de l'objet qui rend le service.

- Le type du veilleur est défini par une interface Java

Les interfaces

Une interface est en Java une déclaration de type qui permet de décrire une capacité.

```
public interface ActionListener {
        public void actionPerformed(ActionEvent evt) ;
}
-------------------// autre exemple dans un autre fichier ".java"
public interface FocusListener {
        public void focusGained(FocusEvent evt) ;
        public void focusLost(FocusEvent evt) ;
}
```

Une classe pourra déclarer adhérer au contrat défini par l'interface. Elle sera obligée d'implanter le code correspondant aux méthodes déclarées mais cela lui permettra d'être "vue" comme conforme au contrat de type exigé.

```
public class Veilleur extends X implements ActionListener {
        ... // codes divers
        public void actionPerformed(ActionEvent act) {
                  // code spécifique
        }
}
```

On pourra alors écrire :

```
monBouton.addActionListener(monVeilleur) ;
```

puisqu'un `Veilleur` remplit les obligations d'un `ActionListener`.

Pour un point sur les interfaces Voir *Plan général d'une interface*, page 313. et les détails concernant les déclarations de méthodes et de constantes de classe dans le chapitre "Rappels: syntaxe, spécifications simplifiées", page 311

Exemple de mise en place de traitement d'événement

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

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

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

public class TestButton {
   public static void main (String args[]){
      JFrame fr = new JFrame ("Test");
      JButton bt = new JButton("Appuyer!");
      bt.addActionListener(new VeilleurBouton());
      fr.getContentPane().add(bt,BorderLayout.CENTER);
      fr.pack();
      fr.show();
   }
}
```

La classe `VeilleurBouton` définit une instance de traitement de l'événement .

```
 import java.awt.event.*;
public class VeilleurBouton implements
                              ActionListener{
   public void actionPerformed(ActionEvent evt) {
      System.err.println("Aïe!sur " + evt.getActionCommand());

      /* faire System.err.println dans un interface graphique n'est
       * pas fondamentalement une bonne idée! ceci est un exemple
       */
   }
}
```

Exemple de mise en place de traitement d'événement

- La classe `JButton` hérite de `AbstractButton` une méthode `addActionListener(ActionListener)`.
- L'interface `ActionListener` définit une seule méthode , `actionPerformed` qui recevra un `ActionEvent`.
- Lorsqu'un objet de la classe `JButton` est créé, on peut enregistrer un objet veilleur pour les `ActionEvent` par l'intermédiaire de la méthode `addActionListener`, on passe en paramètre un objet d'une classe qui "implante" (*implements*) l'interface `ActionListener`.
- Lorsque l'on clique sur l'objet Bouton avec la souris, un `ActionEvent` est envoyé à chaque `ActionListener` enregistré et la méthode `actionPerformed (ActionEvent)`est invoquée.

Remarquons que l'on aurait pu enregistrer plusieurs veilleurs distincts d'événements Action auprès du bouton. On aurait pu également enregistrer un veilleur qui sache traiter plusieurs types d'événement (par exemple Action + Focus); dans ce cas la déclaration de la classe aurait été :

```
public class VeilleurBouton implements
                            ActionListener, FocusListener {
        public void actionPerformed(ActionEvent evt) {
                  ....
        }

        public void focusGained(FocusEvent evt) {
                  ...
        }

        public void focusLost(FocusEvent evt) {
                  ...
        }
}
```

Catégories d'événements

Il existe de nombreux types d'interfaces de veille dans les packages `java.awt.event` et `javax.swing.event`. Quelques exemples AWT:

Catégorie	Interface	Methodes
Action	ActionListener	`actionPerformed(ActionEvent)`
Item	ItemListener	`itemStateChanged(ItemEvent)`
Mouse Motion	MouseMotionListener	`mouseDragged(MouseEvent)` `mouseMoved(MouseEvent)`

Mouse	MouseListener	`mousePressed(MouseEvent)`
		`mouseReleased(MouseEvent)`
		`mouseEntered(MouseEvent)`
		`mouseExited(MouseEvent)`
		`mouseClicked(MouseEvent)`

Key	KeyListener	`keyPressed(KeyEvent)`
		`keyReleased(KeyEvent)`
		`keyTyped(KeyEvent)`

Focus	FocusListener	`focusGained(FocusEvent)` `focusLost(FocusEvent)`

Window	WindowListener	`windowClosing(WindowEvent)`
		`windowOpened(WindowEvent)`
		`windowIconified(WindowEvent)`
		`windowDeiconified(WindowEvent)`
		`windowClosed(WindowEvent)`
		`windowActivated(WindowEvent)`
		`windowDeactivated(WindowEvent)`

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>`, ...)

Dans certains cas le package fournit une classe `XXXAdapter` qui implante toutes les méthodes du contrat de veille mais sans comportement associé. Cette facilité permet de définir des classes de veille simples par héritage de l'`Adapter` et ainsi permet d'éviter d'implanter toutes les méthodes du `Listener` -on ne définit que celles dont on a besoin-.

Point intermédiaire : *Mise en place d'un "veilleur"*

Mini-exercice :

- Reprendre l'exercice de mise en place d'un formulaire question-réponse ("Exercice :", page 38).
  Faire en sorte que lorsque l'on a validé une question dans le champ de saisie, on affiche un écho de cette question dans le composant réponse.
  Précision: la production de la réponse doit être déclenchée quand on "appuie" sur le bouton `JButton` ET quand on "valide" le champs de saisie `JTextField` (en appuyant sur la touche <ENTER>). Quel est le type d'événement qui permet de gérer ces deux actions de la même manière?

Considérations architecturales: classes interdépendantes

La mise en place d'une interface utilisateur est particulièrement intéressante par les problèmes d'organisation qui sont posés. Il faut faire collaborer des codes qui traitent la logique applicative avec les codes chargés de la présentation. De plus, il faut impliquer les codes qui gèrent les événements et permettent la mise à jour des instances .

Dans le cadre de techniques d'organisation, on utilise souvent des modèles architecturaux (*pattern*) qui distinguent le Modèle (les données qui vont être manipulées) de la Vue (la représentation graphique de ces données) et mettent en place des automatismes de mise à jour.

Sans entrer dans ces techniques nous allons nous poser un problème d'organisation courant: imaginons un exemple très simple dans lequel un `JFrame` contient un bouton `JButton` et un `JTextField` dans lequel nous affichons le nombre de clics sur le bouton.

Si nous écrivons un code tel que :

```
bouton.addActionListener(new ControleurAction());
```

Comment l'instance de contrôle (de type `ControleurAction`) peut-elle connaître le champ de type `JTextField` pour lui demander d'afficher un message?

Considérations architecturales

Le gestionnaire d'événement dans une classe externe

Le fait que le contrôleur doive connaître un composant sur lequel il doit agir nous amène à réaliser une classe qui garde en mémoire une référence sur ce composant cible :

```
public class ControleurAction implements ActionListener {
        private int compteur ;
        private JTextField message ;

        public ControleurAction (JTextField cible) {
            message = cible ;
        }

        public void actionPerformed(ActionEvent evt) {
            message.setText("nombre de clics =" + (++compteur));
        }
}
```

et dans le code de disposition de composants graphiques :

```
bouton.addActionListener(new ControleurAction(leTextField));
```

Quelques remarques sur cette conception:

On écrit un code de classe *ad hoc* : il n'est pratiquement pas réutilisable en dehors de ce besoin précis. De plus la maintenance n'est pas facilitée: dans le code de la classe d'interaction on n'a pas le code de contrôle d'événement "sous les yeux", une modification de la réalisation peut nécessiter une intervention sur deux fichiers. Que se passe-t-il si on décide de remplacer le champ `JTextField` par un `JLabel`?
Cette remarque devient d'autant plus pertinente qu'on peut être amené à ne créer qu'une seule instance de `ControleurAction` et à cibler de nombreux composants dans le code de gestion d'événement.

Considérations architecturales

Le gestionnaire d'événement intégré

Le gestionnaire d'événement peut être intégré à la classe de disposition graphique:

```
public class Disposition extends JFrame implements ActionListener{
        ...
        private JTextField leTextField ;
        ....
        private int compteur ;

        public void actionPerformed(ActionEvent evt) {
         leTextField.setText("nombre de clics =" + (++compteur));
        }

        ....
           bouton.addActionListener(this) ;
        ...
}
```

Ici on a tout "sous la main" mais:

- si on doit veiller sur plusieurs composants (par exemple faire de cette manière des `addActionListener` sur plusieurs boutons) le code du gestionnaire d'événement risque de devenir scabreux -il devra implanter des comportements différents en fonction du composant *source*: ce qui est criticable dans le cadre d'une programmation "à objets"-

- si, de plus, le code doit assurer simultanément la veille de Listeners complexes comme `MouseListener` et `WindowListener`, on risque d'aboutir à un fouillis de gestionnaires d'événements.

Considérations architecturales

Le gestionnaire d'événement dans une classe interne

Une autre solution consiste à définir une classe de veille dans le contexte de la classe de disposition. Ceci est rendu possible en Java par la définition d'une **classe interne**.

```
public class Disposition extends JFrame {
        ...
        private JTextField leTextField ;
        ....
        private class ControleurAction implements ActionListener{
            int compteur ;
            public void actionPerformed(ActionEvent evt) {
                  leTextField.setText(
                          "nombre de clics =" + (++compteur));
            }
        }// fin classe interne

        ....
           bouton.addActionListener(new ControleurAction()) ;
        ...
}
```

On a ici une classe membre d'instance de la classe `Disposition` qui est située dans le contexte de la classe englobante. Du coup l'instance du veilleur a directement accès au champ "leTextField".

Considérations architecturales

Le gestionnaire d'événement dans une classe anonyme

Dans l'exemple précédent on a une classe interne qui est définie alors qu'il n'y a qu'une seule instance utilisable. Il est donc possible d'utiliser une écriture plus synthétique qui est celle d'une **classe anonyme** :

```
public class Disposition extends JFrame {
        ...
        private JTextField leTextField ;
        ....
           bouton.addActionListener(new ActionListener() {
              int compteur ;
              public void actionPerformed(ActionEvent evt) {
                  leTextField.setText(
                          "nombre de clics =" + (++compteur));
              }
            }) ;// fin classe anonyme
        ...
}
```

La syntaxe indique une définition à la volée d'une classe après l'appel du `new`. Cette classe (sans nom) fait implicitement `extends Object implements ActionListener` .

En voici un autre exemple (une fenêtre AWT qui ne dispose pas de l'otion DISPOSE_ON_CLOSE de `JFrame`)

```
fen.addWindowListener( new WindowAdapter() {
        public void windowClosing(WindowEvent evt) {
                dispose() ; //méthode de Frame
        }
}) ;
```

Ici la classe anonyme fait implicitement `extends WindowAdapter` (on a considéré que l'on était "dans" le code d'une classe dérivée de `Frame`, sinon il aurait fallu écrire `fenêtre.dispose()`)

Exercices : Mise en place d'un "veilleur"

Exercices :

- Reprendre l'exercice précédent de mise en place d'un formulaire question-réponse et réorganiser le code de manière à le rendre le plus réutilisable possible.

- Mise en application : faire en sorte que les saisies soient de la forme:

```
cos 0.234
sin 0.254
....
```

(donc en général: `fonction valeur_double`) afficher le résultat du calcul (`java.lang.Math`).

Pour analyser la chaîne utiliser un `java.util.StringTokenizer`. Pour réaliser la transformation chaîne->double voir classe `java.lang.Double`.

Processus légers (threads) 3

Points essentiels:

Comment faire réaliser plusieurs tâches en même temps? En confiant ces tâches à des “processus” différents. Java dispose d’un mécanisme de “processsus 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

Multi-tâches (rappel)

Comment un système d'exploitation permet-il d'avoir plusieurs tâches qui s'exécutent en même temps? Même s'il ne dispose que d'un seul processeur pour réellement exécuter une tâche, il est capable de donner une "illusion" de parallélisme: pendant qu'il pilote une imprimante, il continue à dialoguer avec l'utilisateur par exemple.

Dans un système d'exploitation multi-tâches il y a plusieurs programmes (processus) qui "tournent" en même temps. Chacun exécute un certain code et dispose de ses données propres. L'illusion de l'exécution en parallèle est obtenue en n'exécutant une partie de code de chaque processus que pendant un laps de temps très court. Si chaque tâche ne voit à son tour qu'une partie de son code exécutée on aura l'impression que plusieurs programmes s'exécutent en parrallèle.

A l'interieur du système l'**ordonnanceur de tâches** a un rôle essentiel : c'est lui qui décide quel sera le processus actif à un moment donné et pour combien de temps.

Pour des raisons diverses il existe des **algorithmes d'ordonnancement** différents selon les systèmes : un algorithme de *time-slicing* alloue des tranches de temps pratiquement équivalentes pour chaque processus, dans d'autres algorithmes un processus actif garde "la main" jusqu'à ce qu'il se bloque sur un appel système comme une opération d'E/S.

Ces algorithmes sont complexes car ils tiennent compte de facteurs supplémentaires comme des priorités entre processus.

Processus légers: principes

Dans de nombreux systèmes d'exploitation modernes la notion de tâche est gérée à un niveau plus fin. A l'intérieur même d'un processus il existe des processus "légers" qui partagent les mêmes données. Un même "programme" peut donc faire vivre plusieurs tâches en son sein, et, le cas échéant, le système est capable de ventiler ces tâches sur plusieurs processeurs.

Java intègre la notion de *thread* à l'intérieur même du langage. Une fois créée cette tâche va disposer d'une pile d'exécution qui lui est propre et partager des codes et des données des classes.

La J.V.M. d'exécution pourra soit associer un *thread* Java à un *thread* système soit elle-même réaliser un algorithme d'ordonnancement.

Pour permettre une grande souplesse d'adaptation la spécification ne place que très peu de contraintes standard sur les comportements d'un *thread*. Le programmeur ne doit donc surtout pas faire d'hypothèses sur le comportement de l'algorithme d'ordonnancement.

Voici un exemple de mauvaise conception:

```
// A PROSCRIRE !!!
Image img = getToolkit().getImage(urlImage) ;//dans un composant AWT
    //  méthode asynchrone: le chargement est réalisé par une tâche de fond
while(-1 == img.getHeight(this)) { /* rien : polling */ }
    // tant que l'image n'est pas chargée sa hauteur est -1
```

Ce programme (à proscrire!) est susceptible de "tomber en marche" sur des systèmes utilisant le *time-slicing* et ne fonctionne absolument pas sur d'autres (tout en étouffant la C.P.U.!)[1].

1. L'attente du chargement complet d'une image (si elle est nécessaire) se fait à l'aide d'un objet `MediaTracker`.

Conception d'un Thread

On peut considérer que la classe Java `Thread` fournit un "moteur" qui permet d'exécuter une tâche. Une instance de `Thread` va disposer de caractéristiques propres comme un nom permettant de l'identifier ou une priorité, mais surtout elle va fournir une pile d'exécution pour dérouler un code.

Le code exécuté par le `Thread` est fourni par une autre classe qui réalise le contrat d'interface `Runnable` au travers de sa méthode `run()`.

```
import java.io.* ;

public class Compteur implements Runnable {
        public final int max ;
        public Compteur( int max){
                this.max = max ;
        }

        public void run () {
                for(int ix = 0 ; ix < max ; ix++) {
                   try {
                     File temp = File.createTempFile("cpt","");
                     System.out.println("#"+this+":"+temp);
                     temp.delete() ;
                   } catch (IOException exc){/*test */}
                }
        }// FIN DE TACHE
}
```

Ceci nous permet la définition d'une tâche comme :

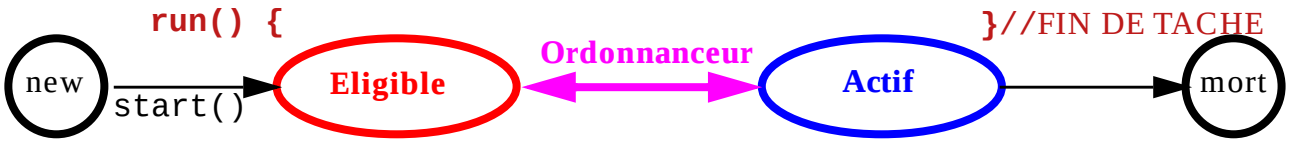
```
Thread tâche = new Thread(new Compteur(33)) ;
```

Ici il s'agit d'une tâche de démonstration sans portée pratique (la "formule magique" dans la boucle crée des fichiers temporaires), mais le fait de faire un appel système lourd dans la boucle va probablement mettre en lumière le comportement de l'ordonnanceur.

Cycle de vie : début et fin de vie

Une fois créé le `Thread` est prêt à être activé: ceci se fait par l'invocation de la méthode `start()`sur l'instance

```
tâche.start() ;
```



A partir de ce moment le code du `run()` est pris en charge par l'ordonnanceur. En fonction des décisions de l'ordonnanceur le `Thread` passe par une série d'états: par exemple *actif* (le code "tourne") ou *éligible* (le code ne "tourne" pas, mais l'ordonnanceur peut le réactiver à tout moment).

Quand le code du `run()` est épuisé (on arrive à l'accolade fermante de la méthode) le `Thread` devient "zombie": l'instance du `Thread` existe encore (jusqu'à ce qu'elle soit éventuellement récupérée par le *garbage-collector)* mais on ne peut plus lui demander de re-exécuter la tâche (`IllegalThreadStateException`).

Pour tester si un Thread est "vivant" :

```
if( tâche.isAlive()) { ....
```

Cette méthode indique que `start()` a été exécutée et que l'instance cible n'est pas devenue "zombie".
Cette méthode n'indique pas que le `Thread tâche` est en état actif (c'est le `Thread` demandeur qui exécute le code qui est actif!).

Point intermédiaire : lancement de threads

Mini-exercice :

- Reprendre le `Thread Compteur` décrit précédemment. Créer un `main` qui lance plusieurs `Compteurs` en parallèle et observer l'ordonnancement (quel *thread* est actif à quel moment).

Cycle de vie: retrait de l'état actif

Dans le code exéctué par un *thread* il est possible de demander à passer en état d'inéligibilité pendant un certain temps. Le code qui s'exécute va s'interrompre, et le *thread* est retiré du *pool* des *thread*s éligibles par l'ordonnanceur:

```
public class Compteur2 implements Runnable {
        public final int max ;
        public Compteur2( int max){
                this.max = max ;
        }

        public void run () {
                for(int ix = 0 ; ix < max ; ix++) {
                        System.out.println("#"+this+":"+ix);
                        try {
                                Thread.sleep(1000) ;
                        } catch (InterruptedException exc) {
                                ...// message?
                        }
                }
        }// FIN DE TACHE
}
```

La méthode de classe `Thread.sleep(long millis)` rend le *thread* qui l'exécute inéligible pendant **au moins** `millis` millisecondes.
On notera que cette méthode est susceptible de propager une exception si on invoque sur l'instance courante de ce *thread* la méthode `interrupt()`.

La méthode de classe `Thread.yield()` permet au *thread* qui l'exécute de demander à l'ordonnanceur de bien vouloir le faire passer à l'état éligible. Si il existe d'autres *thread*s de priorité au moins équivalente qui sont en attente il est possible que l'ordonnanceur décide de rendre un de ces autres *threads* actif[2].

2. Ceci ne garantit pas que le `Thread` qui exécute `yield` ne sera pas immédiatement reactivé par l'ordonnanceur. Ce dernier est maître de sa stratégie, même s'il y a d'autre *threads* éligibles il peut tout à fait conserver le Thread demandeur en activité puis finalement le désactiver sur un appel ultérieur de `yield()` (ou pour une autre raison).

Cycle de vie: attente de la terminaison d'une tâche

La méthode d'instance `join()` permet au `Thread` qui l'exécute d'attendre la fin de l'exécution du `Thread` cible :

```
public class CompteurChaine extends Compteur{
        private Thread autre ;

        public CompteurChaine( int max, Thread autre){
                super(max) ;
                this.autre = autre ;// gag si on fait un cycle!
        }

        public void run () {
                System.out.println("prêt:" + this ) ;
                try {
                        autre.join() ;
                } catch (InterruptedException exc) {/*test*/}
                super.run() ; // mais oui! rien de magique
        }// FIN DE TACHE
}
```

Le *thread* qui va exécuter ce `Runnable` va attendre la fin de l'exécution du `Thread` passé en paramètre pour exécuter la suite du code.

Autres techniques de création

On peut également définir une classe dérivée de `Thread` qui spécialise sa propre méthode `run()`.Ceci peut se produire lorsqu'on a besoin d'un *`thread`* avec un comportement spécifique (ou aussi pour simplifier l'écriture d'un code):

```
public class UnThread extends Thread {
        public void run() {
                //qqch a exécuter
        }
}
```

Exemple:

```
Thread patience = new Thread() {
          public void run() {
                    while(!isInterrupted()) {
                              System.out.print('.') ;
                    }
          }
   };
patience.start() ;
..... //on fait des tas de choses
patience.interrupt() ;
try {
          patience.join() ;
} catch(InterrruptedException xc) { ...}
```

On notera également deux méthodes intéressantes en phase d'initialisation:

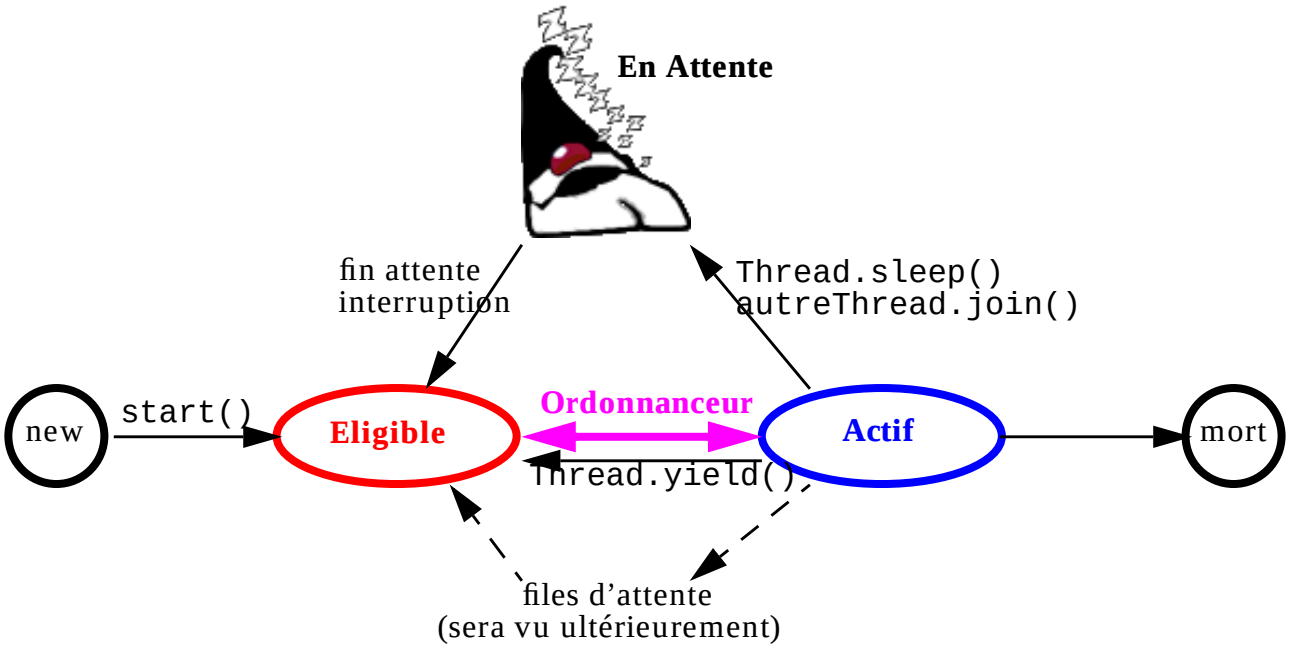
- `setPriority(int  prior)`: permet de fixer la priorité du Thread (une valeur comprise entre les constantes MIN_PRIORITY et MAX_PRIORITY)
- `setDaemon(boolean  b)` : permet de spécifier si le `Thread` est un "daemon" ou un `Thread` "utilisateur". La machine virtuelle Java s'arrête automatiquement quand les seuls *threads* actifs sont des "deamon".

Point intermédiaire : la vie des threads

Mini-exercice :

- Le `Thread` “patience” dans l’exemple précédent ne marche pas de façon complètement satisfaisante. Le modifier de façon à ce qu’il tente d’alterner son exécution avec le `Thread` principal qui l’a créé (pour un test essayer par exemple que l’un affiche “ping” et l’autre “pong” -essayer d’abord avec `yield` puis avec `sleep` -).
  Vos conclusions?

Etats d'un Thread (résumé)



`Exercice` : *threads*

Exercice:

- Concevoir et réaliser un composant Swing `ImageAnimee` :
  un *Thread* permet de déclencher à intervalles réguliers l'affichage successif des images d'un tableau.
  (Rechercher d'abord le composant Swing qui va vous servir de point de départ, chercher dans la documentation comment changer l'image courante affichée. Dans le `main` de test charger les images depuis le système de fichier courant -nous verrons ultérieurement une autre manière de recherhcer ces images-).

Compléments

Les compléments techniques suivants constituent une annexe de référence

ThreadGroup

Les *threads* d'une application peuvent directement interagir entre eux. En pratique on peut limiter ces interactions en partitionnant l'application en groupes de `Thread`s `ThreadGroup`.

Les `ThreadGroup`s sont organisés de manière hiérarchique. Un `Thread` dans un `ThreadGroup` pourra obtenir des informations uniquement sur son groupe (voir méthode `checkAccess()`).

Des méthodes comme `setDaemon(bool)` ou `setMaxPriority(pri)` permettent de fixer des caractéristiques globales au groupe (et eventuellement transmises aux sous-groupes créés). De la même manière l'invocation de `interrupt()` permet de diffuser une demande d'interruption aux `Thread`s et sous-groupes du groupe.

Une possibilité intéressante est de créer une classe dérivée de `ThreadGroup` et de redéfinir sa méthode `uncaughtException(Thread, Throwable)`: on peut ainsi mettre en place un traitement par défaut des exceptions qui remontent la pile et qui n'ont pas été "capturées" (pour un traitement décent Voir *Rapports, journalisations, assertions*, page 141.).

Variables partagées liées à un Thread

Dans certaines situations on peut avoir besoin d'une variable partagée qui ne soit pas liée à une classe (membre `static` de classe) mais liée à un contexte de *thread*. Typiquement une telle variable pourrait être un identifiant de session utilisateur sur un serveur: un ensemble de codes pourraient avoir besoin d'une variable partagée qui soit spécifique au *thread* qui les exécute.

Pour répondre à cette situation on peut mettre en place un objet partagé de type `ThreadLocal`.

```
public class Session {
        public static ThreadLocal uid = new ThreadLocal() ;
        ...
}

public class TacheUtilisateur implements Runnable {
        Utilisateur ut ;
        public TacheUtilisateur(Utilisateur u) {
                ut = u ;
        }
        public void run() {
                Session.uid.set(ut) ;
                ....
        }
}
```

et maintenant dans un code de l'application:

```
utilisateurCourant= (Utilisateur) Session.uid.get();
```

Des *thread*s créés par un `Thread` principal peuvent hériter des valeurs partagées de niveau `Thread` en utilisant un objet `InheritableThreadLocal`.

Images de la mémoire

Le partage de données par plusieurs *thread*s pose des problèmes compliqués (voir blocs `synchronized` dans la leçon suivante). Il faut savoir qu'en Java les variables sont stockées dans une mémoire principale qui contient une copie de référence de la variable. Chaque `Thread` dispose d'une copie de travail des variables sur lesquelles il travaille. Les mises à jour réciproques de ces versions d'une même variable reposent sur des mécanismes complexes qui sont décrits dans le chapitre 17 de JLS (spécification du langage). Il faut comprendre que ce modèle permet des réalisations très performantes en laissant une grande liberté aux compilateurs/exécuteurs et permet au maximum d'utiliser des registres, les instructions en tunnel des processeurs, etc.

Les blocs `synchronized` décrits dans prochaine leçon ont pour effet de mettre en place des barrières qui forcent ces mises à jour. En dehors de ces blocs il est possible de forcer des réconciliations de valeurs en déclarant des variables membres comme `volatile` .

```
public class Partage {
        public volatile int valeurCourante;
}
```

Dans ce cas un `Thread` sait que lorsqu'il "prend" une copie de cette variable sa valeur peut subir des modifications et les compilateurs doivent s'interdire certaines optimisations. Les lectures et écritures de ces variables suivent donc un protocole particulier.

Accès concurrents 4

Points essentiels

Dans la mesure où l'on ne contrôle pas complètement l'ordonnancement 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 `Threads` 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 un file d'attente où il attendra une notification d'un autre `Thread` qui rend la ressource disponible (mécanisme `wait/notify`).

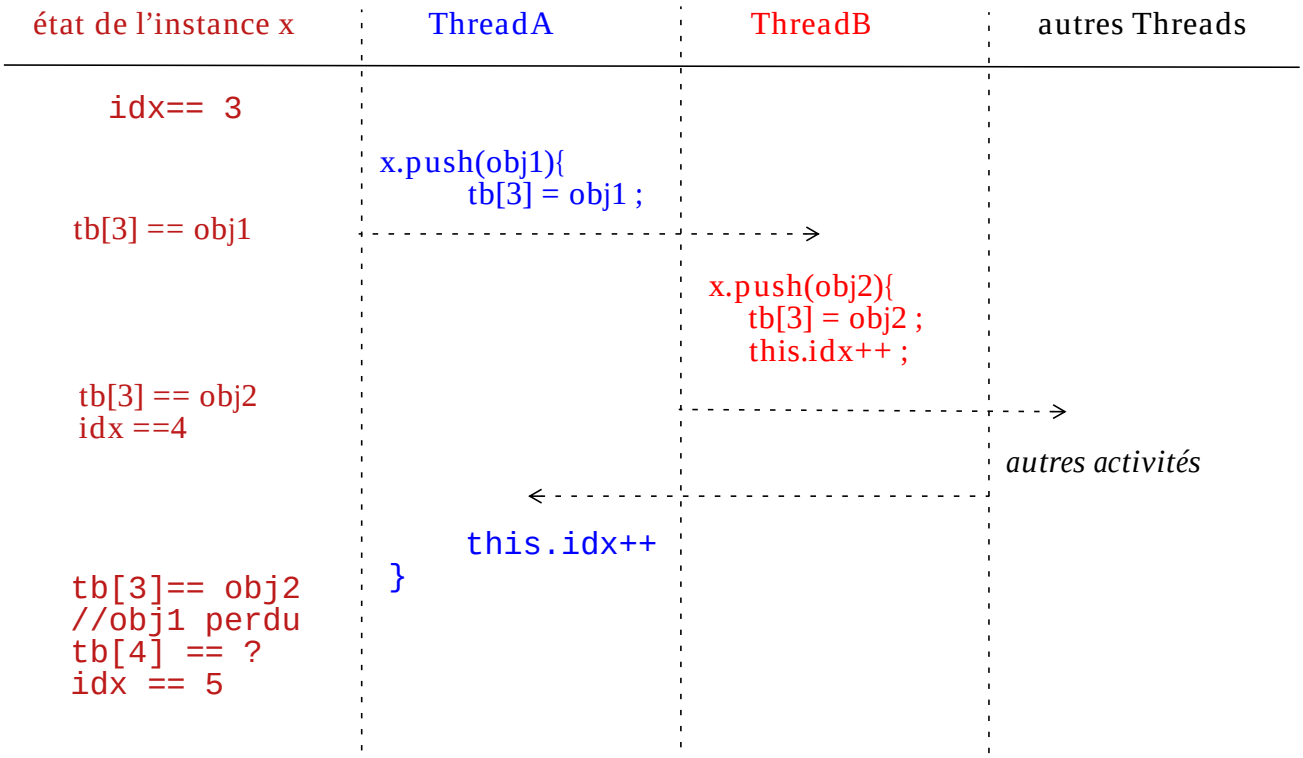
Accès concurrents: problématique

Imaginons une classe gérant une pile d'objets (ceci n'est qu'un exemple: il serait inutile d'écrire un tel code puisqu'une classe existe `java.util.Stack`!)

```
class MaPile {
        ... // code divers dont définition de MAX
        private Object[] tb = new Object[MAX] ;
        private int idx = 0 ;

        public void push(Object obj) {
                //on suppose que l'on traite le pb de la taille
                ...
                tb[idx] = obj ;
                this.idx++ ;
        }

        public Object pop() {
                // on suppose qu'on empêche
                // de dépiler une pile vide (exception)
                ....
                this.idx-- ;
                return tb[idx] ;

                /* autre version avec ménage mémoire
                 *        Object res = tb[idx] ;
                 *        tb[idx] = null ;
                 *        return res ;
                 */
        }
        ....// autre codes dont par ex. la "taille" de la pile
}
```

Notons qu'après chaque appel l'index `idx` indique le premier emplacement disponible dans le tableau pour mettre un nouvel objet.

Et maintenant étudions ce qui se passerait si plusieurs *threads* pouvaient faire des demandes à la même instance de `MaPile` (souvenons nous que nous ne sommes pas maître de l'ordonnancement des *threads*).



Dans ce scénario (et bien d'autres), on arrive à un état incohérent de l'instance!

Ici le problème vient du fait que les deux membres `tb` et `idx` de l'instance participent à un état cohérent : si l'un est modifié il doit être garanti que l'autre est modifié en conséquence. On a ici des blocs de code tels que : lorsqu'un thread a commencé une exécution il faut garantir que le même thread sera seul autorisé à agir dans ce bloc.

Pour traiter ce problème on a mis en place en Java un mécanisme de moniteur.

Blocs `synchronized`

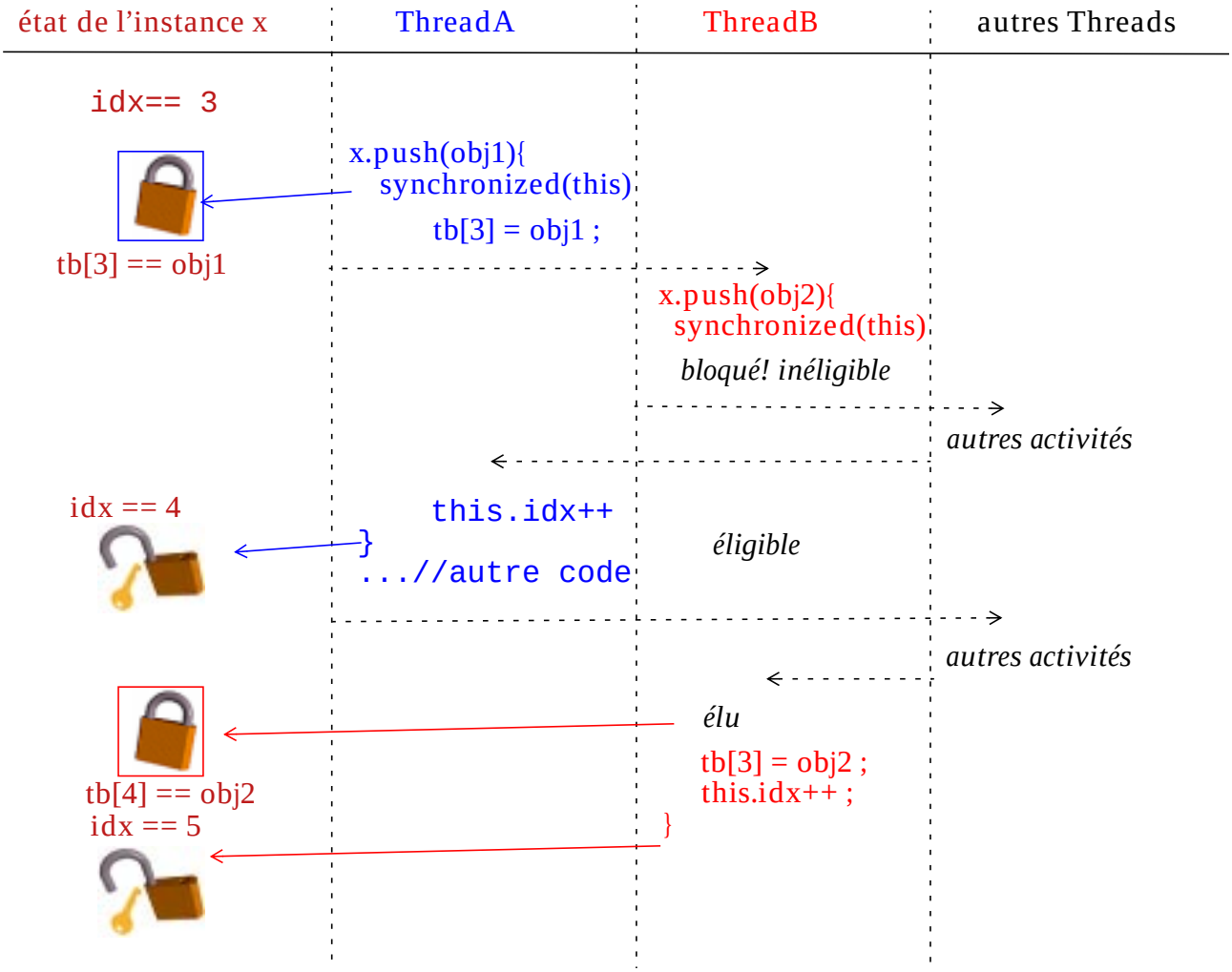
Reprenons partiellement le code précédent et dotons une partie du code d'un bloc qui permette de protéger une partie du code contre des accès concurrents intempestifs:

```
class MaPile {
        ... // code divers dont définition de MAX
        private Object[] tb = new Object[MAX] ;
        private int idx = 0 ;

        public void push(Object obj) {
                //on suppose que l'on traite le pb de la taille
                ...
                synchronized(this){
                        tb[idx] = obj ;
                        this.idx++ ;
                }
        }
```

En Java tout objet est doté d'un **moniteur** : c'est un dispositif qui gère un verrou et des files d'attente d'accès à ce verrou.

Ici le bloc `synchronized` est rattaché à l'objet courant : tout *thread* qui va chercher à entrer dans le bloc devra tenter d'acquérir le verrou ou restera bloqué en file d'attente en attendant son tour d'acquérir le verrou

Blocs `synchronized`



Dans ce scenario:

- Le thread A demande le premier le verrou sur l'instance courante et l'acquiert. Même s'il n'est plus actif il conserve le verrou. Quand il redevient actif il sort du bloc et relache le verrou.

- Le thread B tente d'acquérir le verrou, il passe en file d'attente et ne peut plus être éligible tant qu'il n'est pas sorti de la file. Quant il est élu il acquiert automatiquement le verrou qu'il a demandé.

Autres points sur l'utilisation du moniteur

- Tous les codes qui manipulent le même ensemble de valeurs qui doivent rester cohérentes doivent impérativement se synchroniser sur le même moniteur. Donc pour compléter le code précédent il faut aussi:

```
public Object pop() {
          ....
          synchronized(this) {
                    this.idx-- ;
                    return tb[idx] ;
          }
}
```

- Normallement les blocs `synchronized` doivent être les plus étroits possible et se concentrer juste sur le code critique. Toutefois lorsque toute une méthode est synchronisée sur l'instance courante, il et possible d'adopter la déclaration suivante :

```
public synchronized void push(Object obj) {
          ...
}

public synchronized Object pop() {
          ...
}
```

- Il n'est pas toujours possible de choisir l'instance courante comme support du moniteur:
  - Le code peut contenir plusieurs combinaisons de membres "solidaires":

```
class EtatsMultiples {
        Truc objetX;
        int entierX;
        Object moniteurPourX = new Object() ;
        Machin objetY ;
        int entierY ;
        Object moniteurPourY = new Object() ;
```

- On peut, par exemple, utiliser un objet passé en paramètre d'une méthode (voir réserve ci-après)

```
static Comparable[] trier(Comparable[] tb) {
        synchronized(tb) {
                // suppose que tout utilisateur de ce tableau
                // soit au courant! et fait lui même appel
                // au moniteur avant d'accéder
                ..... // re-arranger les éléments du tableau
                return tb; // sort du bloc synchronized
```

- Java ne détecte pas automatiquement les étreintes mortelles (*deadlock*). Si le code exécuté par un *Thread* A acquiert un moniteur X, puis tente d'acquérir un moniteur Y, si un *Thread* B a acquis Y et tente d'acquérir X tout est bloqué! A moins d'avoir une mauvaise conception il est peu probable que ce code soit écrit dans la classe `EtatsMuliples` ci-dessus, mais pour un moniteur partagé entre plusieurs classes (passé en paramètre par exemple) le risque existe.

- Une méthode de classe peut être `synchronized`: le moniteur utilisé est le champ statique "`.class`" de la classe courante (objet de type `Class`). Dans un code `static` il est bien sûr possible de se synchroniser sur une variable partagée.

- Classiquement un système de verrou est source de problèmes de performance (à la fois pour des raisons liées au déroulement de l'application et des raisons liées aux performances intrinsèques des moniteurs). Il faut donc faire attention. Typiquement les classes de collection du package `java.util` ne sont pas synchronisées : on peut transformer une collection avec `Collections.synchronizedCollection(Collection)`, mais on a aussi des stratégies optimistes qui permettent de parcourir une collection non synchronisée avec un iterateur et d'être averti si une modification intempestive se produit (`ConcurrentModificationException`).

- Pour tester si le `Thread` courant a acquis le verrou sur un objet donné :`Thread.holdsLock(objet)` ; (essentiellement utile pour vérifier des préconditions).

Point intermédiaire : blocs synchronized

Mini-exercice :

- Reprendre l'exercice "pingPong" précédent ("Mini-exercice :", page 70) et faire en sorte que chaque fois qu'un des threads commence à afficher un "Ping" (ou un "Pong") il soit garanti qu'il puisse afficher 5 fois le même message consécutivement.
  (exemple : 5 Pings, 5 Pongs, 5 Pings, 5 Pings, etc...)
  -bien choisir l'objet support de la synchronisation-

Producteur/consommateur: problématique

Imaginons maintenant que nous écrivions le code d'une classe contenant des objets et fonctionnant en file FIFO (*First In First Out*). Nous pourrions ici faire usage d'une des classes `ArrayList` ou `LinkedList` de `java.util`. Ces collections ordonnées ont la particularité d'assurer la contiguité de leurs éléments : on peut toujours rajouter un élément en fin et quand on prend le premier élément de la liste, le second devient le premier, etc.

```
public class FIFO {
        private ArrayList liste = new ArrayList();

        public Object prendre() {
                return liste.get(0) ;
        }

        public void poser(Object obj) {
                liste.add(obj) ;
        }
}
```

Maintenant un objet de cette classe va être partagé par plusieurs *threads*: certains remettent dans la file des objets dont ils n'ont plus besoin, d'autres viennent chercher un objet pour l'utiliser.

Le code de cette classe pose un problème pour le *thread* "preneur": si le stock d'objets est vide, d'une part, la méthode `get` propage une exception (qu'il va falloir traiter) et, d'autre part, il va falloir "attendre" qu'un autre *thread* veuille bien remettre un objet dans le stock. On ne peut pas imaginer une boucle infinie qui teste sans arrêt la taille de la liste (*polling*)!.

Il nous faut ici un mécanisme tel que le *thread* preneur "affamé" (qui se trouve face à une ressource vide) puisse passer dans un état d'inéligibilité qui durera tant qu'un *thread* fournisseur n'a pas signalé explicitement qu'il a effectivement déposé un Objet.

Ici aussi le mécanisme du moniteur va être mis à contribution, un objet (l'instance courante) va servir de support à un dispositif de coordination entre *threads*.

Producteur/consommateur: mise en oeuvre

Reprenons le code précédent pour qu'il assure une coordination entre producteurs et consommateurs:

```
public class FIFO {
        private ArrayList liste = new ArrayList();

        public synchronized Object prendre() {
                while( 0 == liste.size()) {
                        try {
                                this.wait() ;
                        } catch (InterruptedException exc) {
                                // on peut toujours logger!
                        }
                }
                return liste.get(0) ;
        }

        public synchronized void poser(Object obj) {
                this.notify() ;
                liste.add(obj) ;
        }
}
```

Le principe général est le suivant :

- si la liste est vide les *threads* affamés vont se mettre dans un *pool* associé à l'instance courante. La méthode `wait()` met le *thread* qui l'exécute dans ce pool.

- un *thread* qui vient déposer un objet signale que les *thread*s en attente peuvent redevenir éligibles pour aller rechercher un objet dans le stock. Il exécute sur l'objet support du moniteur la méthode `notify()` .

Ce code appelle de nombreuses remarques :

Producteur/consommateur: mise en oeuvre (suite)

```
public class FIFO {
        private ArrayList liste = new ArrayList();

        public synchronized Object prendre() {
                while( 0 == liste.size()) {
                        try {
                                this.wait() ;
                        } catch (InterruptedException exc) {
                                // loguer ou
                                //gérer des interruptions
                        }
                }
                return liste.get(0) ;
        }

        public synchronized void poser(Object obj) {
                this.notify() ;
                liste.add(obj) ;
        }
}
```

- Les méthodes `wait` et `notify` (de `Object`) ne peuvent être appelées que si le *thread* courant a acquis le moniteur sur l'objet concerné. Bien que le compilateur ne vérifie pas cette condition au *runtime* on verrait apparaître une `IllegalMonitorStateException` ("thread not owner").
  Bien entendu le bloc `synchronized` doit englober tout le code "fragile" (ici par exemple il faut être sûr au moment du `get(0)` que la liste n'a pas été modifiée - et vidée!-).
  Pour cette raison faire, dans l'exemple, `notify` avant le `add` n'est pas incorrect!

- `wait()` peut etre interrompu (voir méthode `interrupt()`) et exige donc d'être placé dans un bloc try-catch.

- Le thread qui exécute wait() est mis dans un pool d'attente, quand il en sort il "revient" dans le bloc `synchronized` et entre en compétition avec d'autres threads pour acquérir le verrou courant. Il n'est donc pas certain que la condition qui a accompagnée le `notify` soit encore remplie au moment où le *thread* acquiert enfin le verrou: le test sous forme `while` est **obligatoire**!

- `notify` existe sous deux formes : `notify()` (un *thread* est pris dans le *wait-pool*) et `notifyAll()` (tous les *threads* du *wait-pool* vont chercher à acquérir le verrou). Noter que le choix d'un `Thread` particulier dans le pool est dépendant d'une stratégie de réalisation de la JVM: le programmeur ne doit donc pas faire d'hypothèses à ce sujet.
  Invoquer `notify()` ou `notifyAll()` alors qu'il n'y a aucun *thread* dans le *wait-pool* ne prête pas à conséquence, on peut donc le faire sans risque.

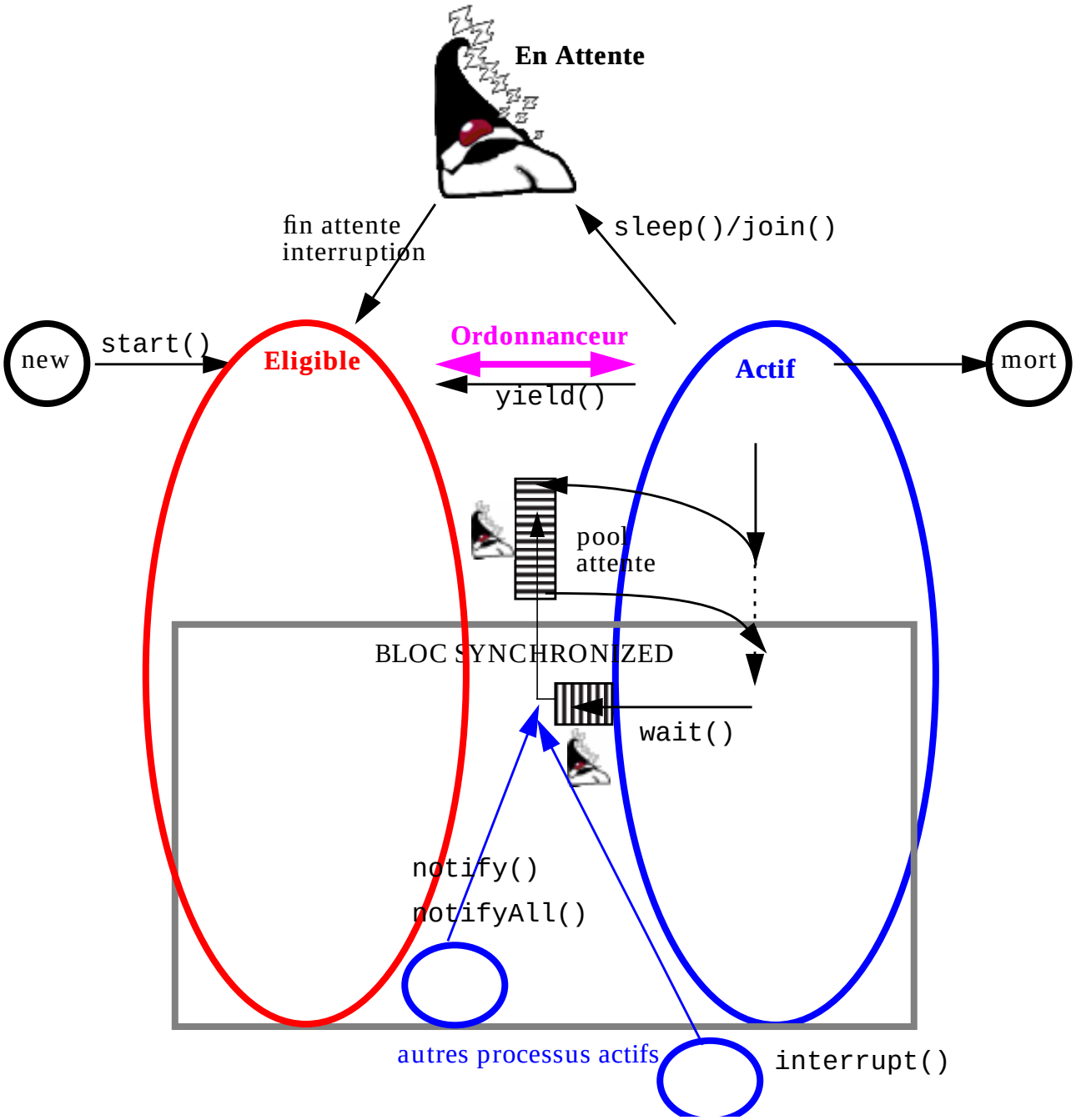
- Il existe une version de `wait` qui permet de limiter le temps d'attente: `wait(long attenteEnMillisecondes)` . Bien entendu à la "sortie" on n'est vraiment pas sûr que la condition attendue soit remplie!

Point intermédiaire : producteur-consommateur

Mini-exercice :

- Reprendre l'exercice précédent (séquence de 5 messages) et s'assurer que les messages vont alterner (5 pings, 5 pongs, etc.). Alternativement chaque processus se mettra en veille pour ensuite attendre que l'autre le "réveille".
  (la solution est très simple, mais n'est pas immédiatement évidente!).

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.

Exercice : threads

Exercice (au choix):

- En utilisant la classe `Fifo` (vue précédemment) concevoir une file spéciale qui contient des Threads (appellons la `ThreadPool`).

  Ces *threads* sont particuliers :

  - leur méthode run() est une boucle infinie
  - en début de boucle le *thread* se met de lui-même dans le `ThreadPool` et se bloque en attente.
  - maintenant lorsqu'on soumet un `Runnable` au `ThreadPool`, Ce dernier retire un thread, le réactive et lui fait exécuter la méthode `run` du `Runnable`. Quand cette tâche est terminée le *thread* se replace en début de boucle. etc.

  (question subsidiaire: à quoi peut servir ce dispositif?)

- Reprendre le composant `ImageAnimee` ("Exercice:", page 72) et faire en sorte que l'on puisse suspendre l'animation, la reprendre et,éventuellement, l'arréter définitivement.

  Lire d'abord "suspend, resume, stop", page 93, on procédera alors de la manière suivante :

  - définir un type de `Thread` spécial appelé `Trame`
  - Cette `Trame` dispose de méthodes qui sont appelées par d'autres *threads* : `suspendRequest()`, `resumeRequest()`, `stopRequest()` . Ces méthodes ont des effets sur l'instance courante qui seront pris en compte par une méthode `going()` qui permet au thread courant de savoir s'il peut continuer à travailler, ou s'il doit se bloquer en attente d'une demande de continuation (ou d'arrêt).
  - Utiliser une instance de `Trame` pour animer `ImageAnimee`.

  (question subsidiaire: cette conception présente un défaut majeur, lequel?).

Compléments

Les compléments techniques suivants constituent une annexe de référence

Exécution des constructeurs

Le code d'un constructeur ne peut pas être `synchronized` : il n'est exécuté que par un seul *thread* pour la création d'un objet donné et, normalement, une référence sur l'objet en construction ne devrait pas être rendue avant la fin de l'exécution du constructeur.

Il reste qu'il y a, potentiellement, quelques situations dangereuses: passer une référence sur l'objet courant dans le code d'un constructeur (enregistrement d'un *callback* par exemple) risque d'offrir un accès à l'instance avant que celle-ci ne soit dans un état cohérent! Le danger vient de l'emploi dans le constructeur de méthodes qui "cachent" le fait qu'elles passent `this` à d'autres objets: l'exemple le plus courant se trouve dans les constructions de hiérarchies (exemple: hiérarchies Container / Component en AWT).

suspend, resume, stop

Dans des versions précédentes de Java il existait des méthodes qui permettait à un `Thread` d'agir "de l'extérieur" sur un autre `Thread.`

Ces méthodes (`suspend()` pour suspendre l'activité, `resume()` pour faire repartir le thread suspendu, `stop()` pour arréter le thread) ont été rendues obsolete (*deprecated*) car le *thread* demandeur ignorait tout de l'action courante du *thread* cible et cela était susceptible de provoquer des catastrophes. Pour plus de détails lire:
`docs/guide/misc/threadPrimitiveDeprecation.html.`

La méthode `interrupt()` reste, elle, intéressante pour réveiller des *thread*s abusivement somnolents.

Deuxième partie: ressources

.

Archives, ressources, sécurité 5

Points essentiels

Le déploiement professionnel d'une application ne saurait se réduire à une simple installation d'un ensemble de répertoires et de fichiers binaires. Nous allons voir quelques techniques simples de déploiement

- archives Jar
- ressources applicatives
- contraintes de sécurité

Les ClassLoaders

Un “programme” java a la particularité d’être un assemblage dynamique de codes. Où la machine virtuelle trouve-t-elle ces codes? :

- Dans les librairies Java du runtime (JRE). Ce sont les librairies “bootstrap” de Java qui font partie de l’installation de la machine virtuelle.

- Dans les “extensions installées” de Java. (a priori des librairies “standard” installées dans le sous-répertoire `jre/lib/ext` de l’installation du SDK)

- Dans les codes “applicatifs” qui sont les codes “vus” par le CLASSPATH au moment du lancement de la machine virtuelle. Dans certaines documentations ces codes sont improprement qualifiés de codes “système”.

Les codes java sont chargés par des `ClassLoader`s. Les `ClassLoader`s ont une organisation hiérarchique : une classe reconnue par le `ClassLoader` “père” est toujours chargée en priorité ; comme le `ClassLoader` primordial (*bootstrap*) est père de tous les autres on ne peut donc pas charger une classe qui porterait le même nom qu’une classe Java fondamentale.

En cours d’exécution d’autres `ClassLoader`s peuvent être créés pour mettre en oeuvre des techniques de chargement spécifiques (par exemple pour charger des codes au travers du réseau), et pour créer des espaces de gestion de classes liées à une “origine” de code (sécurité).

La méthode statique `ClassLoader.getSystemClassLoader()` retourne le ClassLoader “racine” des applications.

Archives jar

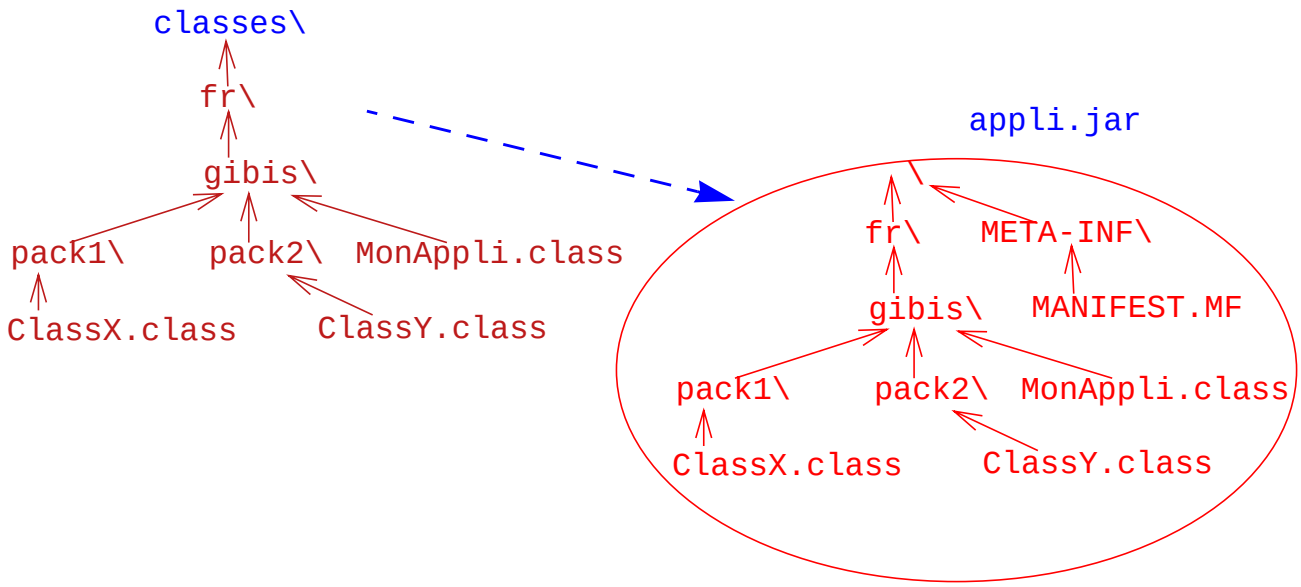
Le ClassLoader applicatif sait retrouver des classes dans une arborescence de fichiers: dans une liste de répertoires qui constituent le `classpath` il cherche un sous-répertoire `fr.acme` pour trouver la classe `fr.acme.Truc`. Pour simplifier notablement la mise en place des librairies qui constituent une application les classes peuvent être déployées dans des archives JAR.

Le lancement d'une application peut alors ressembler à ceci :

```
java -cp /opt/gib/rt.jar:/opt/gib/graph.jar fr.gibis.MonAppli
// l'application utilise 2 archives pour ses classes
```

Une archive Jar est un sur-ensemble des fichiers ZIP et contient de manière compressée une représentation d'une arborescence de fichiers.
Exemple de fabrication d'une archive jar:

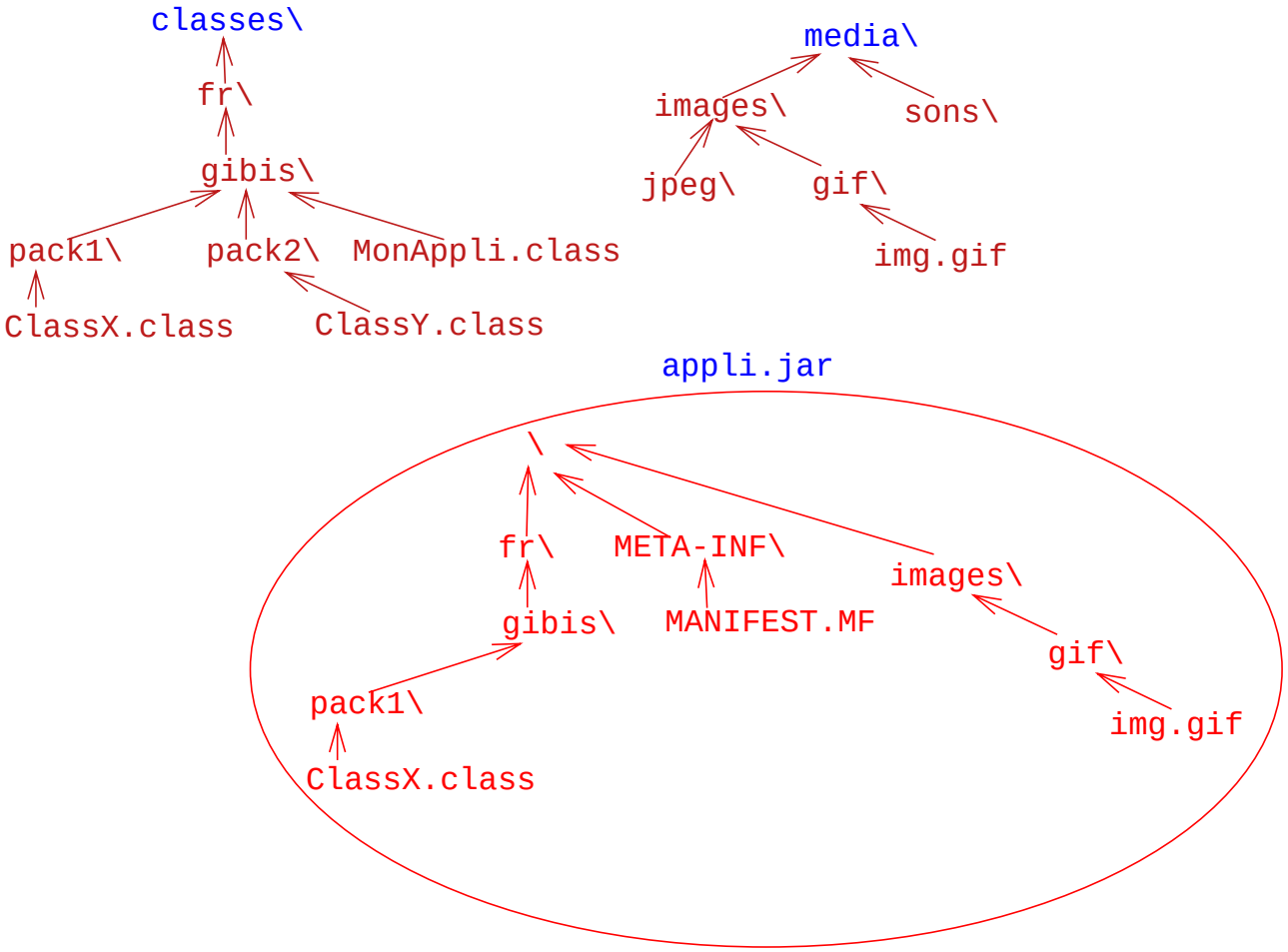
```
jar cvf appli.jar fr
```



Ici la commande a été lancée dans le répertoire "classes", l'archive jar a créé en son sein un "répertoire" supplémentaire (META-INF) qui contient un fichier "manifeste" contenant des meta-informations sur le contenu de l'archive.

On peut aussi recomposer une archive à partir des plusieurs parties d'arborescence situées dans des répertoires différents:

```
jar cvf appli.jar -C classes fr/gibis/pack1 -C media images/gif
```



Pour lister le contenu d'un jar :

```
jar tvf appli.jar
```

Pour compléter un jar existant (*update*):

```
jar uvf appli.jar images/jpeg
```

Archives jar: lancement d'une application

L'utilisation d'archive jar simplifie le déploiement, et permet aussi d'améliorer les performances de chargement: si des codes de classes sont distants -comme dans le cas d'une applet- les classes sont chargées en une seule opération. 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.gibis.MonAppli

```

(noter la ligne vide en fin de fichier -obligatoire-)

Création du fichier jar :

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

Lancement de l'application :

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

Dans de nombreux systèmes de fenêtrage un simple "double-click" sur l'icône d'un fichier jar permet de démarrer l'application (à condition que le champ `Main-Class` ait été renseigné)

Point intermédiaire : pratique des archives Jar

Mini-exercice :

- Reprendre une des applications graphiques précédentes utilisant des packages et plusieurs classes. Fabriquer une archive Jar

- Lancer l'application depuis cette archive.

Ressources

A l'exécution un programme Java peut avoir besoin d'autres fichiers que des fichiers ".class", par exemple:

- des fichiers images (ou son!)
- des fichiers texte (messages, configuration, ...)
- etc....

On qualifie globalement ces fichiers de **ressources** . Les ressources sont nécessaires à l'exécution et sont liées à des codes qui les exploitent. En termes de déploiement il faut donc faire en sorte qu'elles soient présentes et que les codes sachent les rechercher.

Il faut savoir distinguer les ressources propres à l'environnement (liées à la JVM, au site local, à l'utilisateur) et les ressources liées au programme lui-même.

- Ces distinctions ne sont pas évidentes car un programme peut avoir besoin de ressources de configuration, de paramétrisation dont le principe est défini par un programme, mais qui sont fournies par l'environnement (par ex. générées ou décrites localement et installées indépendamment du Jar principal de l'application).
  Voici par exemple un lancement d'une application dans lequel on passe des informations aux classes système:

```
java -Djava.security.policy=sec.policy -jar appli.jar
```

(nous allons voir ultérieurement les fichiers de configuration de sécurité qui sont utilisés dans cet exemple: ici le fichier est situé dans le répertoire de lancement de l'application).
Un autre exemple pourrait être la paramétrisation d'une `Applet` au moyen des balises `param` et d'une méthode comme `getDocumentBase()`.

Ressources applicatives

Dans leur principe les `ClassLoader`s peuvent accéder aux classes par des moyens très divers (y compris au travers du réseau) il est donc naturel qu'ils aient emprunté au monde internet la notion d'URL (Uniform Resource Locator). Un `ClassLoader` standard `java.net.URLClassLoader` fait très spécifiquement référence à ce concept. Or le mécanisme qui sert à rechercher une classe peut aussi être utilisé pour des ressources applicatives liées aux classes qui les utilisent. D'où une notion importante : si une ressource est déployée avec une classe on peut la référencer par une URL déduite de celle de la classe.

```
import java.net.URL;
import java.awt.* ;
import javax.swing.* ;
public class PanelImage extends JPanel {
        static Class maClasse = PanelImage.class ;
        // au runtime : this.getClass()

        public PanelImage(String nomImage) {
                super(new BorderLayout()) ;
                URL url = maClasse.getResource(nomImage) ;
                if(null != url) {
                        add(new JLabel(new ImageIcon(url)),
                                BorderLayout.EAST);
                }
        }
}
```

D'autres méthodes ou constructeurs de Java utilisent des URLs: `getImage()` (de `Applet` et de `java.awt.Toolkit`), méthodes d'`AudioClip` de `Applet`, de `javax.imageio.ImageIO`, ... La technique la plus générale est celle des `InputStream`s (obtenus par `getResourceAsStream()` -voir leçon sur les flots d'entrée/sortie-.

On notera que lorsqu'une classe du *bootstrap* `ClassLoader` demande une ressource elle va, en fait, faire appel au `ClassLoader` "racine" au travers de `ClassLoader.getSystemResource()`.

Point intermédiaire : ressources applicatives

Mini-exercices :

- Fabriquer un Jar comprenant un “sous-répertoire” `images` à la racine (ce répertoire contenant une image).
  Utiliser une de ces images dans une des applications graphiques précédentes..
  Attention: la formulation de l’argument de `getResource()` n’est pas triviale -bien lire la documentation-

- Pour les très rapides (ou ceux qui révisent) :

  - Reprendre l’exercice mais imaginer que les images ne sont pas installées dans le jar des classes, mais sont dans un autre jar “fabriqué” sur le site au moment du déploiement. Que feriez-vous? Expérimentez. Quelles sont les limitations de sécurité? (à voir après avoir suivi la suite du chapitre)

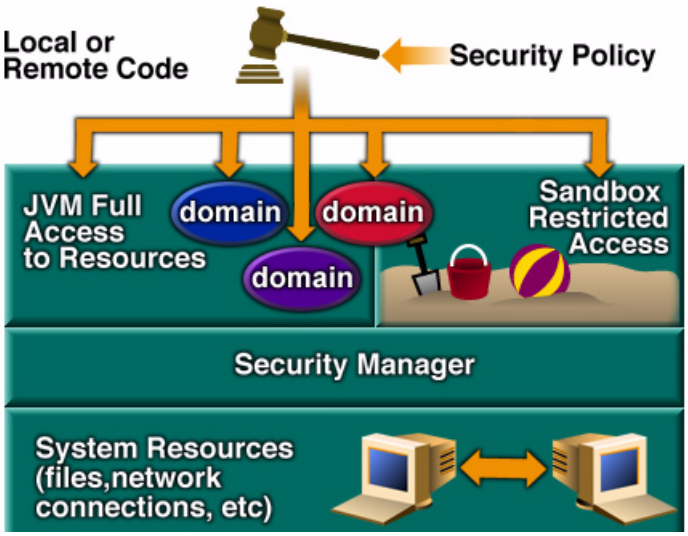
  - (*pour ceux qui connaissent le principe des Properties*) Même problématique pour lire un fichier de “propriétés” par la méthode `load(InputStream)` de `java.util.Properties`.

Sécurité

A partir d'une application on peut mettre en place des `ClassLoader`s chargeant des classes à distance au travers du réseau[1]. Ces codes posent toutefois un problème important: quelle confiance peut-on leur faire? *A priori* aucune:

- Le ClassLoader doit vérifier le pseudo-code
- Les codes qui s'exécutent ont des droits limités en ce qui concerne leur interaction avec l'environnement local, on leur applique une politique de sécurité par défaut nommée *Sandbox Security Policy*. Ces classes ne peuvent:
  - 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 du code (ainsi une `Applet` peut ouvrir une connexion avec son hôte d'origine).

A partir de Java 2 on a défini un moyen standard de gérer des politiques personnalisées en fonction de l'URL d'origine des codes.



1. La technique d'utilisation d'un tel ClassLoader ne sera pas abordée ici. Nous utiliserons de tels dispositifs de manière transparente(Applets, RMI,...)

Sécurité: politique configurée

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 le fichier ".java.policy" de son répertoire d'accueil:

```
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)

Sécurité: politique configurée

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

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

La sécurité distingue les classes fiables (propriété **`java.sys.class.path`**) des autres classes soumises aux contrôles de sécurité (propriété **`java.class.path`**).

Dès qu'une application met en oeuvre un `SecurityManager` (directement ou à travers la mise en place d'un chargement à distance) 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
```

Point intermédiaire : configuration sécurité

Mini-exercices :

- Sur les bases de l'interface-graphique "Question/Réponse" ("Exercices :", page 59) réaliser un code qui tente d'afficher une propriété "système" (`System.getProperty`). Le tester sous `SecurityManager` avec des propriétés critiques comme la propriété "`user.home`" de l'utilisateur (capter la `SecurityException`!). Mettre en place une configuration de sécurité qui permette cet affichage. Utiliser:

```
permission java.util.PropertyPermission "user.home", "read" ;
```

Dans le cas d'une Applet pour "passer" un fichier spécifique de configuration de sécurité au logiciel de test `appletviewer`:

```
appletviewer -J-Djava.security.policy=my.policy MyApplet.html
```

-

Compléments

Les compléments techniques suivants constituent une annexe de référence

Autres services sur les archives Jar

Voir la documentation `<install_java>/docs/guide/jar/jar.html` et en particulier :

- Description dans le Manifeste de packages “complets” (attribut `Sealed:`): toute classe du package marqué “Sealed” sera alors nécesssairement contenue dans le jar courant. Ceci permet de bien gérer la consistance d’une distribution et de sa version et d’améliorer les contraintes de sécurité,

- Description dans le Manifeste des informations de version du package. Ces informations peuvent être éventuellement exploitées par programme, pour s’assurer des compatibilités de comportement. (rappel: en Java le package est l’unité indivisible de livraison et de version des APIs -versions vues du point de vue “utilisateur” de l’API-).

- Mise en place d’un `class-path` auxiliaire des classes dépendantes du jar courant et mise en place d’un index d’optimisation (`jarindex`) pour optimiser le chargement des archives jar dépendantes du jar courant.

Manipulations des certificats de sécurité

Voici un exemple simple de création et de gestion des certificats de sécurité :

Soit une Applet nommée `SnoopApplet.java` et qui lit la propriété “user.home” de l’utilisateur de l’Applet. Normalement la consultation de cette valeur est interdite, on va donc créer une ressource de sécurité (fichier `.java.policy`) qui autorise un serveur particulier (celui de la société `acme.fr`) à réaliser cette opération.

Le serveur va adresser un fichier jar signé au client. Cette signature s’opère avec la clef privé de la société ACME. Par ailleurs ACME met à la disposition du client sa “Clef publique” qui permettra au client d’authentifier l’archive jar.

Le serveur met à la disposition du client un “Certificat” qui contient les informations nécessaires. Ce certificat est stocké localement dans une ressource (`keystore`). L’obtention du certificat peut se faire de diverses manières (par ex. transaction avec une tierce partie qui se porte garante du serveur).

Génération du certificat (coté serveur) :

Ici on supposera que c'est le serveur lui-même qui fabrique le certificat , il va générer ses clefs publiques/clefs privés et va stocker le résultat dans une base locale (keystore), ce certificat est connu sous l'alias "acmeserver":

```
$ keytool -genkey -alias acmeserver
Enter keyStore password : 14juillet
What is your first and last name?
  [Unknown]: Jean Dupond
What is the name of your organizational unit?
  [Unknown] : service informatique
What is the name of your organization?
  [Unknown]: ACME France
What is the name of your City or Locality?
  [Unknown]: Toulouse
What is the name of your State or Province?
  [Unknown]: Midi-Pyrenees
What is the two-letter country code for this unit?
  [Unknown]: FR
Is <CN=Jean Dupond, OU=service informatique, O=ACME France,
L=Toulouse,ST=Midi-Pyrenees, C=FR>
correct?
  [no]:  yes
Enter key password for <acmeserver>
        (RETURN if same as keystore password) acme14
```

On notera que l'accès à la base et au certificat sont protégés par des mots de passe.

Consultation de la base des certificats (coté serveur):

```
$ keytool -list
Enter keystore password:  14juillet

Keystore type: jks
Keystore provider: SUN

Your keystore contains 1 entry:

acmeserver, Tue Mar 09 17:21:07 CET 1999, keyEntry,
Certificate fingerprint (MD5):
ED:55:F3:0D:E4:A1:13:A3:BE:34:AB:7D:DE:BF:C8:B3
```

Signature de l'archive jar (coté serveur) :

```
$ jarsigner snoop.jar acmeserver
Enter Passphrase for keystore: 14juillet
Enter key password for acmeserver: acme14
```

Contenu de l'archive jar :

Dans le répertoire META-INF de l'archive on trouve les fichiers :

MANIFEST.MF

```
Manifest-Version: 1.0
Created-By: 1.2 (Sun Microsystems Inc.)

Name: sl275/secur/SnoopApplet.class
SHA1-Digest: wrvJn8RFHRxB5RilIcYNdNRFCSM=
```

ACMESERV.SF

```
Signature-Version: 1.0
SHA1-Digest-Manifest: 1gAPmlinU6BRogZ5VF0smNXfAVg=
Created-By: 1.2 (Sun Microsystems Inc.)

Name: sl275/secur/SnoopApplet.class
SHA1-Digest: +QoJQIt14oTI8v+AmbD4X7q1BIE=
```

ACMESERV.DSA (données)

Exportation d'un certificat dans un fichier (fournisseur de certificat) :

```
$ keytool -export -alias acmeserver -file acmeserver.certificate
Enter keystore password:  14juillet
Certificate stored in file <acmeserver.certificate>
```

Importation d'un certificat depuis un fichier (coté client) :

```
$ keytool -import -alias acme -file acmeserver.certificate
Enter keystore password:  client14
Owner: CN=Jean Dupond, OU=Service informatique, O=ACME France,
L=Toulouse,ST=Midi-Pyrenees, C=FR
Issuer: CN=Jean Dupond, OU=Service informatique, O=ACME France,
L=Toulouse,ST=Midi-Pyrenees, C=FR
Serial number: 36e54a23
Valid from: Tue Mar 09 17:19:47 CET 1999
until: Mon Jun 07 18:19:47 CEST 1999
Certificate fingerprints:
         MD5:  ED:55:F3:0D:E4:A1:13:A3:BE:34:AB:7D:DE:BF:C8:B3
         SHA1:
C8:E3:54:FD:2C:BA:76:12:27:97:90:FC:BE:47:90:20:59:82:7A:A5
Trust this certificate? [no]:  yes
Certificate was added to keystore
```

Nota: l'alias local du certificat peut être différent de l'alias original ('acme" au lieu de "acmeserver" dans l'exemple). La base et ses mots de passe est bien sûr différente.

Contenu de la base des certificats (Coté client) :

```
$ keytool -list
Enter keystore password:  client14

Keystore type: jks
Keystore provider: SUN

Your keystore contains 1 entry:

acme, Wed Mar 10 17:44:47 CET 1999, trustedCertEntry,
Certificate fingerprint (MD5):
ED:55:F3:0D:E4:A1:13:A3:BE:34:AB:7D:DE:BF:C8:B3
```

Politique de sécurité (coté client) :

Fichier .java.policy

```
keystore "file:/home/leclient/.keystore" ;

grant signedBy "acme" {
 permission java.util.PropertyPermission "user.home","read";
} ;
```

L'utilitaire `policytool` peut permettre de spécifier cette politique de sécurité.

On notera que le fichier décrit l'emplacement de la base des certificats (avec une URL).

Toute classe authentifiée située dans l'archive jar dont la signature correspond à celle désignée par l'alias "acme" aura la possibilité de provoquer une lecture de la propriété "user.home".

Autres aspects de la sécurité

De manière générale les aspects sécurité sont approfondis dans notre cours SL-303. Quelques points toutefois:

- Les permissions accordés à un code au moment de l'éxécution dépendent des permissions accordés aux autres codes appelés dans la pile d'exécution. Ainsi un code relativement privilégié ne peut accéder à des services qui sont refusés à son code appelant. Dans certains cas ce code peut accorder au code appelant des privilèges qui lui sont attachés : voir documentation de java `docs/guide/security/doprivileged.html`

- Les configuration de sécurité abordés ici concernent les droits accordés à des codes; pour des droits accordés à des utilisateurs voir JAAS (`docs/guide/security/jaas/tutorials/index.html`)

Internationalisation 6

Points essentiels

L'internationalisation est un cas particulier d'exploitation des ressources. Java dispose d'un ensemble de mécanismes prédéfinis :

- Définition d'un "contexte culturel": `java.util.Locale`
- Mise en place d'un mécanisme abstrait de recherche de ressources: `ResourceBundle`
- Classe de "formattage" (mise en forme, analyse) dans le cadre de conventions dépendantes de la culture locale.

Pourquoi “internationaliser” une application?

Pourquoi internationaliser :

- Traduire des messages / des interactions
  (pouvoir dynamiquement d’adapter à la langue de l’utilisateur)

- Traduire des messages paramétrés “ ce répertoire contient 1000 fichier(s)”

- Analyser / afficher selon des conventions locales : formats de nombres, dates, formats valeurs monétaires

- Savoir s’adapter à des ordres alphabétiques différents, comparaisons de chaînes.

- Adapter automatiquement l’ordre de disposition de composants
  (exemple: formulaires en arabe `TextField` de saisie situé à gauche du `Label` d’invite -document lu de droite à gauche-)

Une application doit pouvoir s’adapter automatiquement à la langue de l’utilisateur (ou même permettre des écrans multilingues)

Contextes culturels : Locale

Les mécanismes de localisation du language Java utilisent une information de “contexte culturel” définie au moyen de la classe `java.util.Locale`.

Locale : constructeurs, constantes

```
Locale(langue, pays)
Locale(langue, pays,variant)
new Locale(“fr”, “CA”) ; //français du Canada
new Locale (“fr”,”CA”, “WINDOWS”) ;
new Locale (“fr”,”FR”, “EURO”) ;
          // monnaie euro et français de France
```

Les arguments “langue” (chaîne en minuscule) et “pays” (chaîne en majuscules) obéissent à des standard de désignation (voir documentation et méthodes statiques getISOLanguages, getISOCountries).

La classe fournit également des constantes prédéfinies :

```
// constantes de classe
Locale.FRENCH ; // Locale(“fr”,””)
Locale.FRANCE ; // Locale (“fr”, “FR”)
```

Les objets Locale (et les désignations associées comme la chaîne “fr_FR”) permettront de créer des hiérarchies de ressources:

1. Le “Francais”
2. Le “Français” de Belgique
3. La belgique francophone avec l’Euro comme monnaie

Contexte culturel: obtention, mise en place

Au moment où une JVM démarre elle met en place un `Locale` par défaut (en fonction de paramètres de l'environnement)

```
Locale local = Locale.getDefault(); // local

System.out.println(local.toString()) ;
        //-> fr_FR : chaîne 'clef"
System.out.println(local.getDisplayName());
        // Français (France)
System.out.println(Locale.ENGLISH.getDisplayName());
        // Anglais
        // on a traduit en français le nom!
```

On peut positionner globalement cette valeur (`Locale.setDefault(locale)`), mais il est plus pertinent de la fixer pour chacun des objets qui utilisent ces contextes d'internationalisation:

```
NumberFormat nbf = NumberFormat.getInstance(local) ;
        // sert à mettre en forme des nombres
        // on crée un "formatteur" propre au contexte
```

Comment demander à une classe les `Locales` qu'elle sait gérer:

```
Locale[] locs = NumberFormat.getAvailableLocales();
```

On peut agir sur le Locale par défaut au moment du lancement de la JVM en modifiant les propriétés `user.language`, `user.country`, `user.variant`.
Exemple: `java -Duser.language=fr monpackage.MonProgramme`

Paramétrage de ressources: Resourcebundle

La “traduction” (d’un message par exemple) doit s’appuyer sur des ressources extérieures de paramétrage des applications.

Un mécanisme général est defíni pour rechercher ces ressources de localisation: les `ResourceBundle`. Quand on recherche l’équivalent localisé d’un objet on s’adresse à un mécanisme de `ResourceBundle` défini autour d’un “thème”.

- Pour un thème donné on a conceptuellement un dictionnaire qui associe à une clef (unique) une valeur correspondante

- Le résultat de la recherche est en fait obtenu par la consultation d’une hiérarchie de dictionnaires suffixés par les chaînes caractéristiques des Locales. Ainsi par exemple pour un thème libellé “Erreurs” on pourra avoir :

```
Erreurs_fr_CA_UNIX // locale demandé
Erreurs_fr_CA
Erreurs_fr
Erreurs_<Locale_par_défaut>// obtenu par getDefault()
Erreurs_<Locale_par_défaut_simplifié>
Erreurs
```

Une hiérachie ne doit jamais avoir de maillon manquant (si *fr_CH* existe *fr* et <rien> doivent exister) - on peut aussi construire des chaînages explicites (`setParent()`)-

Un dictionnaire abstrait est obtenu par spécification du “thème” et du Locale.

```
ResourceBundle messagesErreur =
        ResourceBundle.getBundle(“Erreurs” ,locale) ;
        // -> Exception MissingResourceException
```

La clef de recherche est une chaine. Selon les circonstances on pourra utiliser:

```
messagesErreur.getObject(clef) // pourquoi pas un glyphe?
messagesErreur.getStringArray(clef)// ou un ensemble de chaines
messagesErreur.getString(clef) ;// ou, bien sûr, une chaîne
```

ListResourceBundle

ResourceBundle est une classe abstraite qui peut connaître des réalisations diverses. Deux systèmes de spécifications de ressources sont livrés en standard : `ListResourceBundle` et `PropertyResourceBundle`.

Ces deux classes sont une illustration des priorités de recherche de getBundle() : à un niveau de la hiérarchie des `Locale` la méthode recherche d'abord une classe portant le nom du niveau par exemple `Erreurs_fr.class` (cette classe peut être de type `ListResourceBundle`) et ensuite un fichier `Erreurs_fr.properties` (fichier permettant de construire dynamiquement une instance de `PropertyResourceBundle`). Ces ressources doivent se trouver dans le chemin d'accès aux classes (classpath) associé à un ClassLoader donné.

Exemple de ListResourceBundle :

```
public class Erreurs_fr extends ListResourceBundle {

   public static final Object[][] tb = {
      { "err" , new ImageIcon("erreur.gif")} ,
      { "ERR" , new ImageIcon("terreur.gif")} ,
   } ;

   public Object[][] getContents() { return tb ;}
}
```

La réalisation doit implanter la méthode `getContents()` et le tableau retourné doit associer une clef (chaîne de caractère) et un objet.

Dans l'exemple ci-dessus seul getObject() devra être utilisé.

PropertyResourceBundle

Un `PropertyResourceBundle` est un Bundle créé automatiquement par exploitation d'un fichier ".properties"

Le format d'un tel fichier est défini pour la classe `java.util.Properties` (voir méthode `store()` ou `load()` ;

```
# Errors_fr.properties
#Supposed to be translated in French ;-)
you\ stupid\!=\ A\u00EDe\! une erreur malencontreuse est survenue
I'll\ scream\!=patience, impossible n'est pas fran\u00E7ais
```

exploitation :

```
ResourceBundle errs = ResourceBundle.getBundle("Errors") ;
...
String message = errs.getString("you stupid!") ;
        // et, heureusement, "fr" est le contexte par défaut
```

Point intermédiaire : Mise en place d'un Bundle

Mini-exercice :

- Reprendre l'interface graphique "Question/Réponse" ("Exercices :", page 59) et modifier la de manière à ce que le message qui annonce une erreur tienne compte de la langue de l'utilisateur (il ne s'agit pas de traduire le message généré par l'exception, mais uniquement le message annonçant qu'il y a une erreur).
  Tester en modifiant le `Locale` par des propriétés passées au lancement de la machine virtuelle. Mettre les ressources de traduction dans un jar constitutif du CLASSPATH.

Classes de mise en forme/analyse

Les objets de mise en forme en fonction du contexte culturel dérivent de la classe `java.text.Format` et disposent de méthodes de mise en forme du type : `String format(Object)` et de méthodes d'analyse de chaînes de caractères du type: `Object parseObject(String)` (et méthodes spécialisées dérivées).

Classes principales de mise en forme : **`NumberFormat`**, **`DateFormat`**, **`MessageFormat`**.

Utilisation de formats numériques prédéfinis :

```
NumberFormat formatteur =
                    NumberFormat.getNumberInstance(Locale.FRANCE);
System.out.println(formatteur.format(345987.246));
```

Donne:

```
345 987,246
```

`NumberFormat` permet aussi de se procurer des instances mettant en forme des montants monétaires (avec mention de la monnaie locale) et des pourcentages (voir aussi la classe `Currency`).

Analyse:

```
NumberFormat formch =
          NumberFormat.getNumberInstance(new Locale("fr","CH"));
try{
             Number nb = formch.parse("345'987.246");
} catch (ParseException exc) {
          ...
```

`parse` tente de retourner un Long ou sinon un Double

Attention le séparateur des milliers en Français est vu comme un "blanc dur" par Java, l'analyseur ne sait pas traiter une espace normale.

Numériques avec instructions de formatage

La classe `DecimalFormat` (sous-classe de `NumberFormat`) permet de controler la mise en forme :

```
NumberFormat formatteur =
                    NumberFormat.getNumberInstance(Locale.FRANCE);
DecimalFormat df = (DecimalFormat)formatteur ;
df.applyPattern("000,000.00") ;
         //séparateurs US + indicateurs chiffres (0 si manquant)
System.out.println(df.format(5987.246));
```

Donne:

```
005 987,25
```

Discussions: quel est l'intérêt d'une méthode "fabrique" comme `NumberFormat.getNumberInstance()` *? Du coup qu'elle est l'anomalie conceptuelle dans le code ci-dessus?*

Autres formats

- `DateFormat` permet d'obtenir des objets de mise en forme/analyse des Dates , `SimpleDateFormat` et `DateFormatSymbols` permettent de fabriquer des formats de mise en forme de Dates.

- `MessageFormat` et `ChoiceFormat` permettent de fabriquer des messages composés de plusieurs éléments paramétrables:

```
int nbfichiers = 10 ;// obtenu en fait par calcul
String nomFic = "prmtc34x5f" ; //;-) idem
String param ="Le disque {1} contient {0} fichier(s)" ;
         // obtenu par exploitation d'un Bundle
MessageFormat fmt = new MessageFormat(param) ;
System.out.println( fmt.format(new Object[] {
                              new Integer(nbfichiers),
                              nomFic ,
                  })); // tableau anonyme (idiome)
```

Donne

```
Le disque prmtc34x5f contient 10 fichier(s)
```

Utilisation d'un paramétrage par exploitation de plages de valeurs (ChoiceFormat)

```
param = "Le disque {1} {0}" ; // obtenu par le Bundle
MessageFormat fmt2 = new MessageFormat(param) ;
double[] limites = {0,1,2} ;
// a chacune de ces limites est associé un format
//{0,number} analyse le parametre qui lui est passé
// (et qui contient seulement un élément de type nombre)
String[] associés =
{"est vide","contient 1 fichier","contient {0,number} fichiers"} ;
ChoiceFormat choix = new ChoiceFormat(limites, associés) ;

// argument 0 de "param" est traité par le ChoiceFormat
fmt2.setFormatByArgumentIndex(0, choix) ;
System.out.println( fmt2.format(new Object[] {
                              new Integer(nbfichiers),
                              nomFic })));
```

La mise en forme donne:

```
Le disque prmtc34x5f contient 10 fichiers
```

Point intermédiaire : saisies/affichages paramétrés

mini-exercice :

- Utilisation de l'interface graphique "Question/Réponse": reprendre le code qui analyse une demande de calcul sin/cos. Faire en sorte que ce code soit internationalisé: l'analyse et l'affichage du format des doubles dépend des convention locales.

Compléments

Les compléments techniques suivants constituent une annexe de référence

Autres point à approfondir

De manière générale c'est le package `java.text` qui traite les problèmes d'internationalisation:

* Pour les comparaisons textuelles voir la classe *`Collator`*

* Autres applications textuelles : pour les limites de lettres et de mots `BreakIterator` (a utiliser pour phrases, lignes, mots, etc.)

* Un certain nombres de composants graphiques (gestionnaires de disposition) peuvent automatiquement tenir compte d'une préférence (voir : Component Orientation)

* Un outil très utile pour internationaliser:
JavaTM Internationalization and Localization ToolKit
(`http://java.sun.com/products/jilkit/`)

A partir d'un certain niveau il est conseillé de faire appel à des spécialistes, car les problèmes peuvent devenir très complexes (exemple : changement d'orientation de lecture à l'intérieur d'un formulaire,...)

Performances

Faire attention aux performances. La recherche d'un `ResourceBundle` par exemple est relativement couteuse: il est donc peu conseillé d'aller en chercher un chaque fois qu'on en a besoin; il est plus prudent d'initialiser cette recherche dans une opération de niveau initialisation de classe qui permette ensuite un accès partagé depuis de nombreuses instances.

Organisation des Bundles

Pour permettre une meilleure gestion de Bundles essayer de lier l'existence des Bundles à celles des packages : par exemple un Bundle par package, rarement plus , (et éventuellement regroupement des ressources communes à plusieurs sous-packages au package "ancêtre" commun). Nous verrons dans un chapitre ultérieur que la politique de gestion des `Bundle`s doit être cohérente avec celle des `Logger`s.

Préférences 7

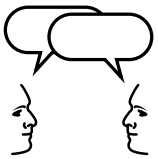
Points essentiels

Les “préférences” sont un cas particulier de ressources: comment laisser l’utilisateur (ou l’administrateur système du site) gérer un contexte de préférences qui soit permanent et, éventuellement, commun à un ensemble d’applications.

Preferences: principes

Les “préférences” d’un utilisateur sont des données rémanentes qui sont utilisées dans des applications pour personnaliser un comportement.

Les applications peuvent permettre à l’utilisateur de les modifier, et, bien entendu, ces données constituent des paramètres comportementaux de ces applications: taille préférée d’une fonte, couleur , etc..



Discussion: que verriez vous comme paramétrages d’application susceptibles d’être traités comme des “préférences”?

Ces paramétrages, tels qu’implantés par le package `java.util.prefs`, ont des caractéristiques particulières:

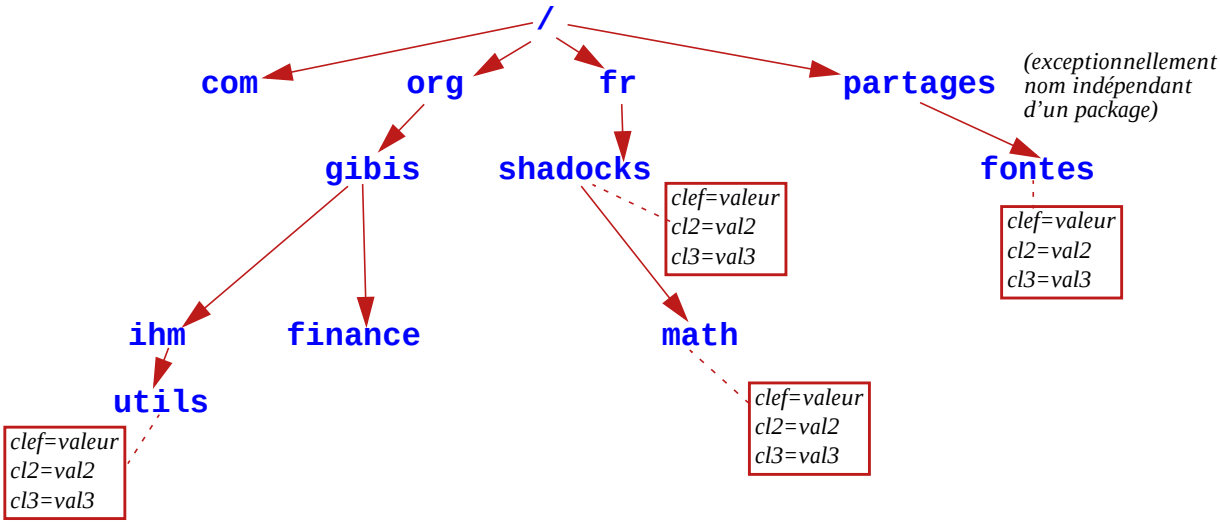
- Ce sont des données non critiques : l’application doit pouvoir fonctionner si elle ne trouve pas le paramètre recherché.
  - Lorsqu’on demande une préférence on doit toujours fournir une valeur par défaut (qui sera renvoyée si la valeur recherchée n’est pas trouvée pour une raison quelconque).
  - Les types possibles pour ces données sont volontairement limités: `boolean`, `int`, `long`, `float`, `double`, `String` et `byte[]`. (voir ci-dessous contraintes du système de persistence).
- Ce sont des données rémanentes qui peuvent aussi être partagées par plusieurs applications:
  - Elles sont stockées de manière transparente à l’utilisateur et de manière spécifique au système (par exemple par défaut dans des registres sous Windows, dans une arborescence de fichiers sous Unix).
  - Il y a une désynchronisation entre les valeurs connues par une instance de J.V.M et les valeurs connues par le système de persistence. Même les modifications sont asynchrones et ne sont pas garanties (seule la demande explicite d’harmonisation avec le stockage sous-jacent est susceptible de propager une erreur si ce stockage est indisponible).

Preferences: contextes de nommage

Spécifier une préférence c'est associer une valeur à une clef. Cette clef de nommage est forcément une chaîne de caractère.

Pour un code il est important de gérer ces associations dans un espace de nommage qu'il contrôle. *A priori* un espace "naturel" est lié au nom du package.

Les préférences d'un utilisateur sont organisées dans un arbre de contextes. L'organisation hiérarchique des contextes de préférences est associée à un arborescence de nommage des "noeuds" (*node*)



Chaque noeud est représenté par une instance de la classe `Preferences`. Sa désignation se faisant par un cheminom absolu ("`/org/gibis/ihm`") ou relatif au noeud courant ("`gibis/ihm`" dans "`/org`").

Attention, pour des raisons de portabilité, il y a une limite à la longueur du "nom" d'un noeud (80!) -comme d'ailleurs à la longueur de la clef ou de la valeur: voir les constantes de la classe `Preferences`-

Il existe deux arborescences de contextes:

- les préférences utilisateur
- les préférences "système"

Obtention d'un contexte

Pour obtenir un contexte dans le cadre d'un code donné:

```
Preferences curPrf= Preferences.userNodeForPackage(MyClass.class);
```

Si le noeud associé au nom du package n'existe pas il est créé (ainsi que tous ses parents manquants), mais ce contexte ne deviendra persistant que si une harmonisation avec le système de stockage devient nécessaire (ou est explicitement demandée). De la même manière il existe une méthode `systemNodeForPackage(Class)`.

On peut obtenir directement la racine d'un des deux arbres de préférences:

```
Preferences racine = Preferences.userRoot();// par exemple
```

A partir d'un noeud on peut obtenir un autre noeud en utilisant son cheminom de désignation:

```
Preferences petitFils = curPrf.node("fils/petitfils");
Preferences absolu = curPrf.node("/fr/shadocks");
```

Dans le premier cas on a obtenu un descendant direct du noeud courant dans le second un noeud arbitrairement choisi dans l'arbre courant.

Consultation dans un contexte

A partir d'un noeud donné on peut consulter une préférence :

```
String imagePréférée = packNode.get("image", "defaultimg.gif") ;
```

Comme pour toute les consultations de valeur on doit donner une valeur par défaut. Certaines implantations de la persistence sont autorisées à conserver cette valeur par défaut (mais ce n'est pas un comportement sur lequel il faut compter -de plus si une "valeur par défaut" stockée existe elle peut remplacer la valeur par défaut proposée!-).

On notera que, du fait de la désynchronisation, la valeur obtenue n'est pas nécessairement celle présente à l'instant *t* dans le système de persistence: la valeur peut-être modifiée par une autre partie de l'application courante (bien entendu les codes des modifications sont `synchronized`) ou alors une autre JVM a concurrement modifié cette valeur dans le système de persistence.

```
package org.gibis.ihm ;
import java.awt.* ;
import java.util.prefs.* ;
....
public class Fontes {
        public static Preferences curPrefs =
                  Preferences.userNodeForPackage(Fontes.class);
        public static final String CLEF_NOM_FONTE = "nomfonte" ;
        public static final String NOM_FONTE_DEFAUT="monospaced";
        .....
        public Font fontePréféréeUtilisateur() {
                  String nom = curPrefs.get(
                            CLEF_NOM_FONTE, NOM_FONTE_DEFAUT) ;
                  int style = curPrefs.getInt(
                            CLEF_STYLE_FONTE, Font.PLAIN);
                  int taille = curPrefs.getInt(
                            CLEF_TAILLE_FONTE, TAILLE_FONTE_DEF);
                  return new Font(nom, style, taille) ;
        }
        ....
```

On aurait pu aussi stocker la représentation de la fonte sous forme d'une chaîne qu'on aurait ensuite analysée
(voir `java.util.StringTokenizer` ou package `java.util.regex`)

Enregistrement d'une valeur

Dans le contexte d'un noeud donné on peut enregistrer une préférence:

```
// dans un interface graphique
String nomImage = myJTextField.getText() ;
packNode.put("image", monImage)
```

On dispose de méthodes pour enregistrer des valeurs ayant les types simples admis pour les références (`putInt`, `putDouble`, `putByteArray`, etc.).

Ici aussi il faut veiller aux problèmes de désynchronisation avec le système de persistence. Il n'est pas certain que la valeur soit immédiatement enregistrée. Pire elle peut ne jamais l'être: si un code n'a pas le droit de modifier un noeud "système" l'exécution risque de ne pas voir passer d'exception (voir ci-après les synchronisations explicites).

```
....
        public static final String LOCALE = "locale" ;
.....
        public void mémoriseContexteCulturel(Locale loc) {
                  curPrefs.put(LOCALE, loc.toString());
        }
```

(ici, typiquement, si on enregistre "`fr_FR`" comme valeur il faudra lors du `get` l'analyser pour reconstituer le Locale).

Une entrée du dictionnaire local peut être supprimée par:

```
curPrefs.remove(nomPropriété) ;
```

Une suppression radicale du dictionnaire par:

```
curPrefs.clear();
```

Harmonisation avec le système de persistence

On peut explicitement demander des interactions avec le système de persistence sous-jacent.

Ici ces invocations sont susceptibles de propager des exceptions de type `BackingStoreException`:

```
curPrefs.flush() ;
```

Sur cet appel le noeud courant et tous ses descendant sont sauvegardés par le système de persistence (l'appel est synchrone: au retour le système est à jour). Si le noeud vient d'être créé le système de persistence peut créer tous les ancêtres nécessaires à son existence.

```
curPrefs.sync() ;
```

Permet d'assurer que les prochaines consultations de valeurs refléterons tous les changements survenus dans le système sous-jacent (mises à jour par d'autres JVM). Comme effet de bord cette méthode assure que les modifications locales sont aussi harmonisées (effet analogue à un `flush()`).

Noter que l'exception `IllegalStateException` punit tout appel sur un noeud qui a été invalidé (par ex. par `removeNode()`).

Noter également qu'un AccessControler qui n'a pas l'autorisation `RuntimePermission("preferences")` va déclencher une `SecurityException` sur toute demande d'obtention de noeud de l'arbre des préférences.

Point intermédiaire : préférences

Mini-exercice :

1. Faire un programme qui fixe un `Locale` de préférence pour l'utilisateur.

2. Récupérer cette préférence dans votre programme "Question / Réponse" internationalisé

Compléments

Les compléments techniques suivants constituent une annexe de référence

Personnalisation d'un système de Préférences

Dans certaines situations on peut être amené à modifier l'implantation de référence de la gestion de préférences. Exemples:

- architecture distribuée pour les "préférences systèmes" qui seraient obtenues à partir d'une ressource centralisée.
- codes destinés à un utilisateur mais qui s'exécutent sur un serveur (voir remarques dans la documentation de `AbstractPreferences`).

Ici il faut fournir des implantations de services répondant aux contrats (*SPI*) définis dans `AbstractPreferences`. Une `PreferenceFactory` doit rendre les arbres utilisateur et système et sa classe doit être désignée dans la propriété `java.util.prefs.PreferenceFactory` .
- pour programmeurs experts uniquement! noter, de plus, que ce dispositif est susceptible de changer dans des versions ultérieures-

Noter également des possibilités d'import/export de préferences via des descriptions XML.

Veille sur des modifications de préférenecs

A l'intérieur d'une JVM il est possible de mettre en place des mécanismes qui permette de prévenir une partie de l'application si l'état d'un noeud a changé de manière significative dans une autre partie. Voir `PreferenceChangeListener` et `NodeChangeListener`.

Rapports, journalisations, assertions 8

Points essentiels

La version 1.4 de Java a apporté une harmonisation des opérations de trace, rapports, etc.

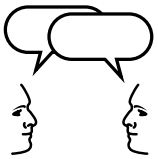
Il fallait ici résoudre une contradiction :

- Il doit y avoir un découplage entre une demande de rapport faite par un composant autonome et la classe qui rend effectivement le sercice.

- Il faut que ce service soit standard pour que des composants indépendants y fassent référence.

Par ailleurs java1.4 a introduit un mécanisme standard d'assertion qui permet de spécifier des vérifications et de faire en sorte que ces vérifications soient activables au lancement d'une JVM.

L'A.P.I. de "log"



Discussion: que faire dans le code d'un bloc `catch` lorsqu'on a récupéré une exception?

Notion de "log"

La notion de "log" recouvre tout ou partie des opérations de traces, mains courantes et reconstitution d'opérations, historiques, signalement d'erreur, monitorat divers, etc.

Dans ces opérations le programme envoie des informations qui concernent des "rôles" très différents: utilisateur de l'application, administrateurs divers, développeurs en phase de tests ou de maintenance, autres applications (analyse, sauvegarde,....), etc.

Une conception saine des opérations de "log" doit suivre les contraintes architecturales inhérentes aux applications Java (modularité, dynamicité):

Exemple: Si au moment de sa conception un composant ne connait pas la nature du contexte dans lequel il va opérer, faire `System.err.println()` d'un rapport d'erreur n'est pas une bonne idée:

- Il peut ne pas y avoir un `System.err` facilement lisible par le destinataire (ex.: Applets, processus "démons" système)
- Il peut être intéressant de différencier les interlocuteurs. Par exemple sur un problème qui trouve sont origine dans le système d'exploitation on pourrait envoyer: un message à l'utilisateur, un message à l'administrateur système, voire un message à la maintenance logicielle.
- Ces messages sont de nature différente selon le rôle de l'interlocuteur: plus detaillé pour l'administrateur, traces détaillées de la pile (`stackTrace`) pour la maintenance etc.

Contraintes architecturales des opérations de “log”

Certains codes ont besoin de poster des informations mais ne sont pas des codes "terminaux" (directement à la surface de l’application comme les codes d’IHM).

Au moment de l’écriture du code on doit :

- pouvoir accéder à un dispositif de log quoiqu’on ne connaisse pas son comportement effectif.
- pouvoir simplement spécifier les informations que l’on loggue et les catégoriser (niveau: trace, info, warning, error).

```
import java.util.logging.* ;
     .....
     // le package est l’unité de base
     Logger logger = Logger.getLogger("org.gibis.monpackage");
     ....
     // Trace dans du code courant
     logger.log(Level.INFO, "!", monObjet)
     .....
     try {
          .....
     } catch (ExceptionStock exc) {
          logger.log(Level.SEVERE, "stock", exc)
// ne remplace pas necessairement toutes les actions a prendre
     }

// nombreuses methodes dérivées  commodes d’acces
     if (TRACE) {//code non compilé si boolén faux au compile time
          logger.entering("CetteClasse", "cetteMéthode") ;
     }
     ....
     logger.warning(
      "ressource manquante..on continue avec valeur par defaut");
     ....
```

On a ici un découplage entre une demande de service et sa réalisation: le code ne connait pas le comportement effectif du `Logger` lié au module applicatif représenté ici par le package.

Point intermédiaire : Loggers par défaut

Mini-exercices :

- Reprendre le code d'analyse et d'évaluation sin/cos ("mini-exercice :", page 128).
  Ne pas renvoyer directement toutes les exceptions vers l'interface graphique mais mettre en place un Logger pour faire un rapport sur certaines exceptions.
  Exécuter le code dans le cadre de l'interface graphique:
  ce comportement par défaut vous satisfait-il? Comment pourrait-on l'améliorer?

Qui détermine un comportement effectif de `Logger`?

Différentes situations et combinaisons sont possibles:

- configuration de déploiement:

  ressources décrivant des comportements et/ou spécifiant des classes à impliquer; classes mises en oeuvre au déploiement.

- outil de contexte :

  ces outils permettent d'exécuter des composants dans un contexte particulier et gèrent de diverses manières le comportement des loggers (y compris dynamiquement).
  Ex.: containers E.J.B., Patrol, etc.

- codes englobant les composants et ayant spécifiquement besoin de mettre en place des comportements:

  Ex.: interfaces graphiques reportant des messages à l'utilisateur

“Réglages” d’un `Logger`

De manière générale on peut dire qu’il y a un Logger par domaine fonctionnel (très souvent associé à la notion de package et portant le nom de ce package).

Sur un Logger on peut effectuer des réglages programmatiques:

- fixer un certain niveau de log et rejeter tout message d’importance inférieure:

```
ceLogger.setLevel(Level.FINE) ;
```

- fixer spécifiquement un filtre programmatique chargé de rejeter certains rapports (classe `LogRecord`).

```
ceLogger.setFilter(monFiltre) // implements Filter
```

- Les rapports sont ensuite propagés à un ou plusieurs `Handler`s qui vont transformer les `LogRecord`s et les mettre en forme pour le monde extérieur.

```
ceLogger.addHandler(monHandlerSurIHM) ;
```

Exemples de `Handler`s prédéfinis : `FileHandler` (log dans un ou plusieurs fichiers), `OutputStreamHandler` (dans un flot d’E/S), `ConsoleHandler` (sur `System.err`), `SocketHandler` (sur des ports TCP distants).

Chaque Handler peut aussi décider:

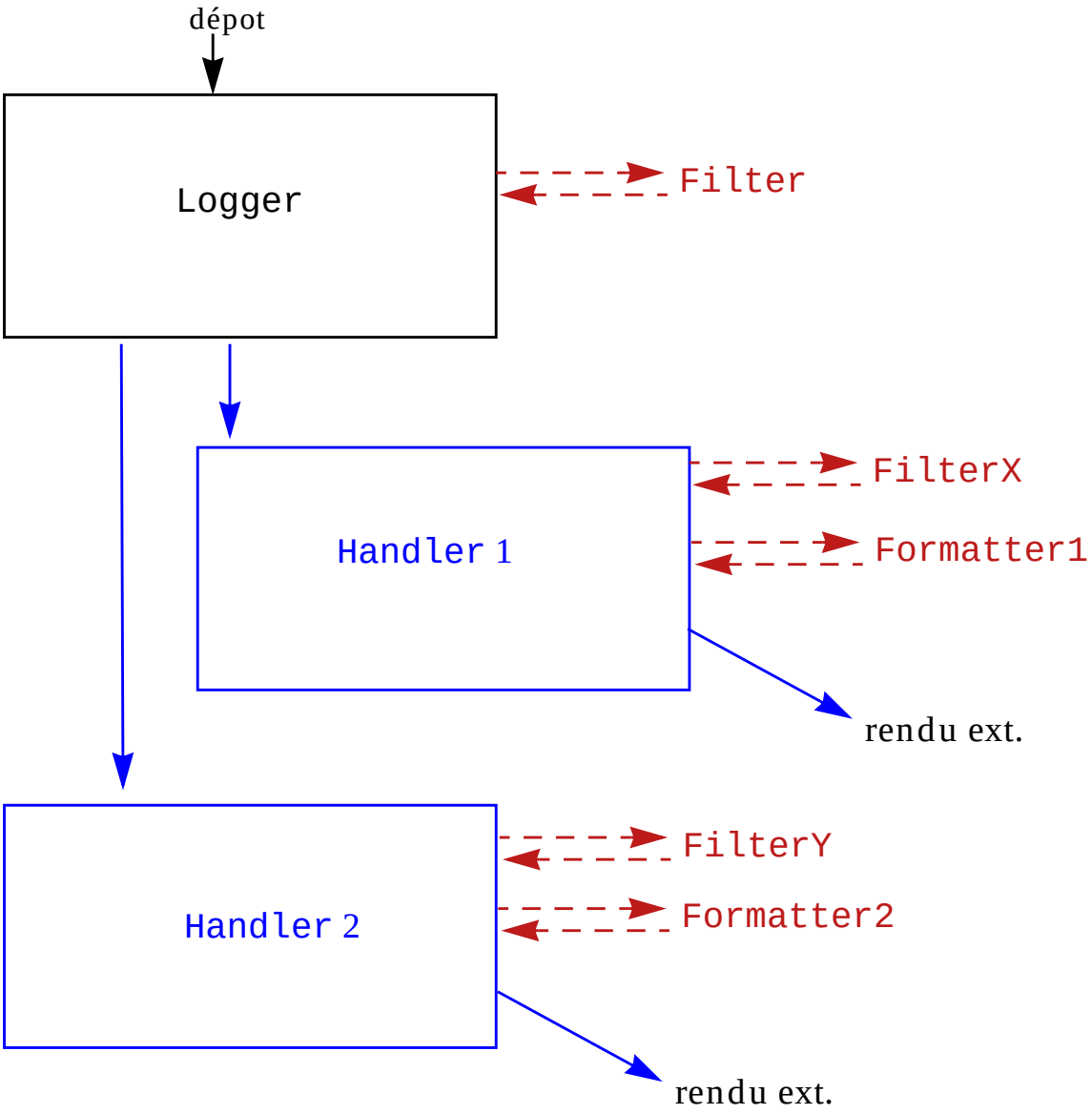
- De filtrer ses messages :

```
monHandlerSurIHM.setFilter(filtre) ;
....
monHandlerSurIHM.setLevel(Level.OFF);// on arrête tout!
```

- de mettre en forme les messages -éventuellement en fonction de la langue (voir package `java.text` et `ResourceBundle`)-:

```
monHandlerSurIHM.setFormatter(new SimpleFormatter());//par défaut
monHandlerSurFichier.setFormatter(new XMLFormatter());
```

Cheminement des rapports



Ce cycle de vie peut être asynchrone (par ex. on sort de la méthode de dépot avant que le message soit écrit effectivement dans un fichier)

Mise en place des `Loggers`

Il serait fastidieux de gérer programmatiquement tous les `Loggers` possibles! En fait il y a une organisation hiérarchique des `Loggers` à partir du `Logger` racine (de nom ““). La hérarchie des Loggers est déterminée par leur nom (`org.gibis` est “père” de `org.gibis.util`) et, par convention, il y a aussi un domaine nommé “`global`”.

Les `Loggers` sont en général créés dynamiquement à la demande et “héritent” du comportement de leur parent:

- Si le niveau de sensibilité (`Level`) d’un `Logger` n’est pas fixé spécifiquement (valeur `null`) il prend automatiquement celui de son ancêtre le plus proche pour lequel ce niveau est fixé.
- Les filtres (`Filter`) sont spécifiques à chaque `Logger`.
- Lorsqu’un `LogRecord` est produit il est soumis à tous les `Handlers` associé spécifiquement au `Logger` et à tous les `Handlers` de ses parents (on peut donc ne pas avoir de `Handler` en propre et hériter de ceux de sa hiérarchie).
  Pour contrer ce comportement:

```
ceLogger.setUseParentHandler(false) ;
```

Au titre d’une configuration globale simplifiée on peut donc agir sur le `Logger` racine (ou sur “global”) . Le système de configuration statique par défaut agit essentiellement à ce niveau (c’est un fichier texte en format `Properties`: voir `logging.properties` et la documentation de la classe `LogManager`).

Point intermédiaire : Configuration statique

Mini-exercices :

- Lire la documentation de `java.util.logging.LogManager` et mettre en place une configuration statique simplifiée pour le `Logger` racine d'une application.

Mise en place explicite d'un `Handler`

Il existe des situations dans lesquelles certains rapports doivent apparaitre explicitement dans l'application elle-même: par exemple si nous voulons faire apparaître dans l'IHM des erreurs "enfouies" qui ne sont pas récupérées dans le code IHM lui-même.

Soit un tel code qui utilise alors pour afficher les rapports un code de `Handler` spécifique de nom `IHMHandler`:

```
Handler monHandlerSurIHM = new IHMHandler() ;
monHandlerSurIHM.setFormatter(new FormatSimple()) ;
Logger.getLogger("").addHandler(monHandlerSurIHM) ;
```

Par ailleurs le code qui émet les rapports le fait en utilisant un `ResourceBundle` particulier :

```
package fr.bankoo.calculs ;

interface PackCst {// portée package!
   ...
   public static final String THEME_BUNDLE = "messages/Errors" ;
   public static Logger CURLOG =
             Logger.getLogger("fr.bankoo.calculs", THEME_BUNDLE);
   ...
```

A partir de ce moment les rapports émis à partir des codes de ce package seront mis en forme en utilisant le *Bundle* correspondant:

```
CURLOG.log(Level.SEVERE,"errBackup",
                new Object[] {deviceName, fileName});
```

Ressource correspondante dans `Errors_fr_FR.properties` :

```
errBackup : sauvegarde de {1} sur {0} impossible
```

Un code de `Handler`

Exemple de `Handler` simple dans le cadre d'un IHM :

```
package xxx ;
....
public class IHMHandler extends Handler {

   private static class IHMDialog extends JDialog {
      // HIDE ON CLOSE par défaut
      private JTextArea jtxt = new JTextArea(10,60) ;
      IHMDialog() {
         // internationaliser
         setTitle("Attention! message!") ;
         getContentPane().add(new JScrollPane(jtxt)) ;
      }
      public synchronized void report(String rapport) {
         jtxt.setText(rapport) ;//p.e. append(rapport)
         pack() ;
         show() ;
      }
   }

   static IHMDialog ihm = new IHMDialog() ;

   public void publish (LogRecord record) {
      //normallement on doit demander une fois
      //                getHead(this) au formatter
      if (isLoggable(record)) {
            ihm.report(getFormatter().format(record));
      }
   }

   public  void flush()  {
   }

   public  void close() {
      //normallement on doit demander une fois
      //                getTail(this) au formatter
   }

}
```

Un code de `Formatter`

Exemple de `Formatter` très simple qui pourra être utilisé avec le code précédent:

```
package xxxx;
import java.util.logging.* ;

public class FormatSimple extends Formatter {
   public String format(LogRecord record) {
      StringBuffer sb = new StringBuffer();
      sb.append(record.getLevel().getLocalizedName())
                .append(":\n\t”)
      // formattage localisé + bundle lié au rapport
               .append(formatMessage(record));
      // exception
      Throwable th = record.getThrown() ;
      if(th != null) {
         sb.append(th) ;
      }
      return sb.toString() ;
   }
}
```

Point intermédiaire : Loggers explicites

Mini-exercices :

- Reprendre le code d'analyse et d'évaluation sin/cos et son interface graphique ("Mini-exercices :", page 144) . Mettre en place les rapports sur interface graphique (en reprenant les codes cités en exemple ici).

Assertions

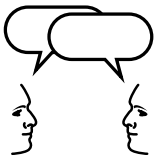
Nous avons vu précédemment des demandes de rapport à l'occasion de traces, de signalement d'erreur etc. Nous abordons ici un thème sensiblement différent qui est celui du déclenchement d'erreurs dans le cadre de vérifications non systématiques.

Java 1.4 a introduit directement au niveau du langage un mécanisme d'assertion au moyen d'un nouveau mot-clef: `assert`.

Une assertion permet de mettre en place un contrôle (par ex. telle valeur est-elle valide?) et ceci de manière non-permanente. Bien souvent ce contrôle est réservé à l'usage exclusif du programmeur ou du metteur au point.

Il est important de distinguer les assertions de l'utilisation "classique" des exceptions. Exemples :

```
package fr.banquoo.comptes;
....
public class Compte {
   private double solde ;
   ....
   public void retrait(double valeur) throws ExceptionDecouvert{
      if(valeur <= 0) {
       throw new IllegalArgumentException("retrait de " + valeur);
      }
      if(valeur > solde) {
         // plutot solde -valeur depasse decouvert autorisé
       throw new ExceptionDecouvert(this, valeur);
      }
      ....
   }
}
```



Discussion: pourquoi avoir ici une exception controlée (ExceptionDecouvert) et une RuntimeException (IllegalArgumentEception)?

Pourquoi utiliser des assertions

Dans l'exemple précédent nous avons un code public qui doit vérifier un certain nombres de préconditions. Le contrôle de l'argument négatif est effectué, il doit être signalé dans la documentation, mais le programmeur utilisateur n'est pas obligé de traiter l'exception correspondante (s'il est sûr que la condition est remplie dans son code d'appel). Soit maintenant le code suivant :

```
package fr.banquoo.comptes;
....
public class Utils {
   static double mensualité(double taux, int durée, double mnt){
         // vérification des préconditions
         ....
   }
```

On a ici affaire à une méthode de portée *package*, qui ne sera utilisée que par les autre classes du même package. Il peut être raisonnable de penser que ces contrôles devront être effectués dans des phases de mise au point, mais qu'il est inutile de les lancer systématiquement dans un code mature. Il est alors possible d'écrire le code suivant :

```
   static double mensualité(double taux, int durée, double mnt){
         assert (taux > 0) && (durée > 0) && (mnt > 0)  ;
         ....
```

On a ici un nouvel élément de syntaxe du langage Java: la compilation doit se faire avec l'option `-source 1.4` (minimum).

Le point intéressant: si l'exécution est lancée avec l'option `-ea` (`enableassertions`) le test sera exécuté et si le résultat est faux une `AssertionError` sera déclenchée. Si les assertions ne sont pas activées le test ne sera pas exécuté.

Pourquoi utiliser des assertions

Pourquoi utiliser des assertions? Pour permettre aux programmeurs de renforcer la robustesse du code en introduisant des tests supplémentaires liés aux phases de développement, déploiement, tests, maintenance,...:

- contrôles de propriétés sur des valeurs (trop souvent laissées implicites).

```
switch(code) {
          case 0 : ....; break;
          case 1 : .... ; break ;
          case 2 : .....; break
          default : // ne doit pas se produire
                    assert false;

}
```

- contrôles sur des invariants de classe, post-conditions (et, dans certains cas particuliers, de préconditions liées à des codes “privés”)

```
static double mensualité(double taux, int durée, double mnt){
   assert (taux > 0) && (durée > 0) && (mnt > 0);
   //précondition spéciale liée à un code de portée limitée
   ....
   assert (res >0) && (doublePrécis(res)) ;// postcondition
   return res ;
}
```

Exemple d’invariant dans `CompteJeune extends Compte`:

```
assert decouvert == 0 ;//pas de decouvert autorisé
```

- erreurs de déploiement, d’assemblage de composants, tests particuliers à certaines conditions d’exploitation:

```
import java.util.Arrays;
.....
Locale loc = ..... ;
NumberFormat frm = NumberFormat.getNumberInstance(loc) ;
frm.format(montant) ;
assert
  Arrays.asList(NumberFormat.getAvailableLocales()).contains(loc);
//ici “format” fonctionne toujours mais dans certains cas
// on voudrait pouvoir savoir pourquoi le format est surprenant
```

Assertions: syntaxe, activations

Syntaxe:

```
assert expressionBooléenne ;
assert expressionBooléenne : expression ;
```

Dans le second cas si les assertions sont activées et si “`expressionBooléenne`” rend `false`, alors “`expression`” est évaluée et son résultat est passé au constructeur d’AssertionError (qui en déduit un `String`).

```
static {
  try {
        bundle = ResourceBundle.getBundle(“Erreurs”, locale) ;
  } catch (MissingResourceException exc) {
        assert false : exc ;
        // déploiement anormal! on arrête tout!!
  }
}
```

L’activation du contrôle des assertions peut être globale, ou spécifique à un package, à une classe :

```
java -ea fr.bankoo.Banque
```

active les assertions dans tous les packages (sauf packages *bootstrap*)

```
java -ea:fr.bankoo... fr.bankoo.Banque
java -ea:fr.bankoo.Banque fr.bankoo.Banque
```

dans un cas on active les assertions uniquement dans les packages et sous-packages de `fr.bankoo` dans l’autre uniquement dans la classe désignée.

On peut également désactiver les assertions pour certaines classes ou packages et intervenir programmatiquement pour demander aux `ClassLoader`s une activation.

Point intermédiaire : codes robustes

Mini-exercice :

- Reprendre vos codes (IHM Question / Réponse, lanceur, évaluateur d'expression mathématiques) et les "blinder" : s'assurer d'une mise en place pertinente d'exceptions d'une part et d'assertions de l'autre.

Discussions: qu'envisageriez-vous de faire pour mettre au point des protocoles de tests systématiques?

Compléments

Les compléments techniques suivants constituent une annexe de référence

Autres point à approfondir

- Les rapports (`LogRecord`) peuvent contenir des objets et sont donc une structure de données potentiellement riche; il faut toutefois être raisonnablement sûr que ces objets seront exploitables (se méfier en particulier de la “serialization” des `LogRecord` -Voir *Dispositifs avancés d’E/S*, page 197.-).
  Noter que la méthode `getThrown()` permet d’exploiter un objet `Throwable` et que depuis la version 1.4 cet objet dispose d’une méthode `getStackTrace()` qui permet d’exploiter programmatiquement les informations de la pile.

- Internationalisation: penser à ne passer par le `Formatter` que le plus tard possible dans la chaine de transfert des rapports. Imaginez ce qui se passerait si vous formattiez sur un serveur un message lu par un utilisateur sur un poste client: il est fort possible que le `Locale` nécessaire ne soit pas pris en compte!

- Les erreurs dans les `Handler`s eux-même ne doivent pas passer par le système de log (voir utilisation de `ErrorManager`).

- Sécurité : naturellement les codes non-fiables ont un accès restreint aux services locaux de log. Voir les autorisations de sécurité et les `Logger`s “anonymes”. Attention: les classes de configuration statique doivent être chargées par le `ClassLoader` primordial.

Questions de performance

Il faut faire extrêmement attention aux opérations de log dans un système critique.

Il faut d'abord réfléchir à la nature des rapports: s'agit-il d'une trace simple ou cette trace est-elle elle même critique pour reconstituer un historique des opérations?. Dans le second cas on a affaire à une opération synchrone importante alors que dans le premier l'opération peut être asynchrone et faire l'objet de mesures d'abandon en cas de détection de problèmes de performances graves. Il ne faut pas que les opérations de log non-critiques deviennent un goulot d'étranglement (imaginez la tempête de logs si ceux-ci participent eux-mêmes à des dégradations du service rendu par l'application!).

Points à surveiller pour les performances du système de log:

- Appels systèmes, E/S non bufferisées, (pour une technique intéressante: voir utilisation de `MemoryHandler`).
- Trop-plein d'informations et problèmes de place (XML particulièrement couteux, mais même `SimpleFormatter` doit être surveillé)
- Recherches complexes (`ResourceBundle`, classes ou programmation dynamiques)
- Blocs `synchronized` (Voir *Accès concurrents*, page 77.)

Troisième partie: Entrées/Sorties

Application d'un "pattern": les flots d'E/S 9

Points essentiels

Ce chapitre a un double objectif :

- présenter le notion de "modèle structurel" (*design pattern*)
- présenter la technique des entrées/sorties en mode flot en montrant que sa réalisation en Java s'appuie sur un modèle de programmation abstrait.

Modèles structurels (design patterns)

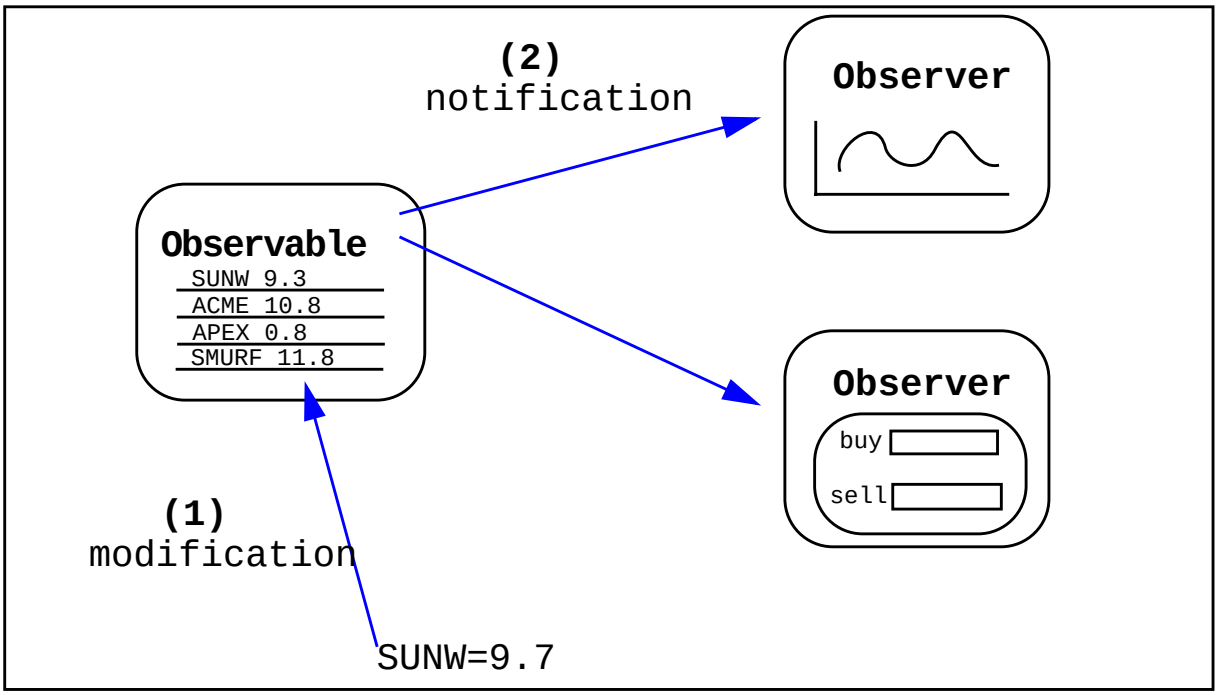
L'expérience accumulée par les programmeurs a permis de constater que certaines classes, bien que traitant des problèmes différents, avaient des analogies structurelles. D'où l'idée de repérer des modèles organisationnels qui permettent de récupérer des mécanismes de fonctionnement pour réaliser des classes ayant des finalités différentes. L'analyse approfondie de la pertinence de ces mécanismes permet en fait de résoudre une classe de problèmes apparentés en ne s'intéressant qu'aux propriétés structurelles des objets mis en oeuvre.

Ces modèles structurels (*patterns*) font l'objet de diverses publications et le programmeur doit les connaître et savoir les adapter à ses problèmes. Un exemple de "petit" pattern en Java est la présence dans le package `java.util` du couple `Observer/Observable`.

Le modèle Observer/Observable

Ce modèle (qui n'est pas sans rappeler le mécanisme des veilleurs d'évenements) est adapté à des situations où différents objets veulent être tenus au courant d'évolutions survenues sur un objet "surveillé" (L'`Observable`). Les objets qui implantent l'interface `Observer` ont pour obligation de s'enregistrer auprès de l'`Observable` et de réagir aux notifications que ce dernier leur enverra en cas de modification de son état.

Une application typique serait d'avoir diverses interfaces graphiques qui "partagent" un tableau de valeurs boursières. Chaque fois qu'une valeur est modifié dans le tableau de référence toutes les interfaces graphiques doivent refléter l'évolution.




```
public class CoursBourse extends Observable {
          private HashMap cours ;
          ....
          // p.e. synchronized
          public void changeCote(String valeur, Money cote){
                    cours.put(valeur,cote) ;
                    setChanged();//rend notification possible
                    notifyObservers(valeur) ;
          }
          ....
}
```

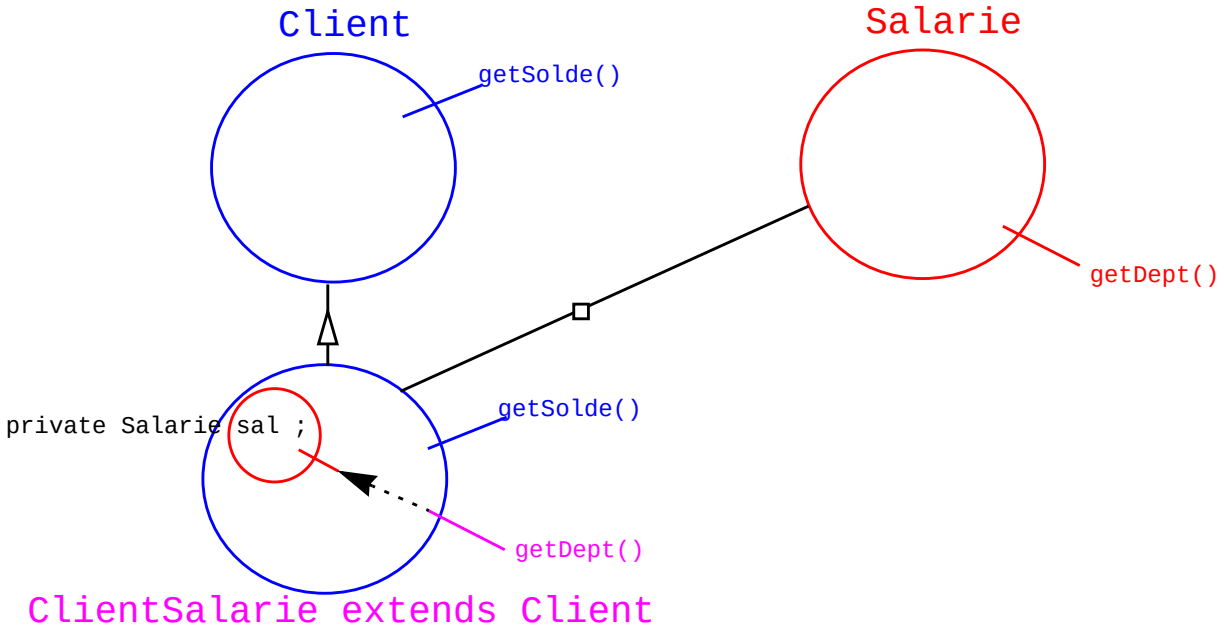
```
public class AfficheCours extends ... implements Observer {
          // A la création s'enregistre auprès de l'Observabl
          ....
          public void update(Observable cours, Object valeur)
                    // Met à jour l'affichage
          }
}
```

Un exemple d'utilisation de "pattern"

Nous allons prendre cet exemple dans le code de Java lui-même. Les dispositifs d'entrée/sortie en mode flot sont construits autour d'une interprétation d'un modèle structurel connu: le "décorateur".

Introduction: combinaison héritage/délégation

Il n'existe pas d'héritage multiple en Java. Si l'on veut récupérer des comportements de deux classes différentes il faut explicitement l'écrire

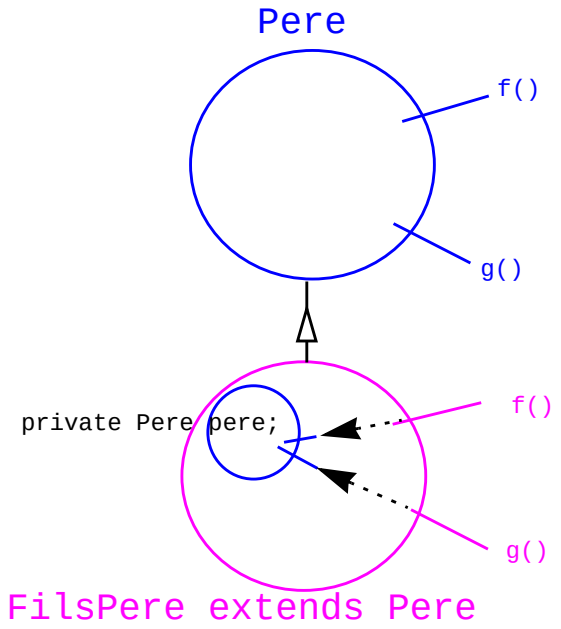


Ici le `ClientSalarie` hérite des méthodes de `Client` et définit des méthodes de `Salarie` en déléguant leur réalisation à une instance locale de `Salarie`.

Très probablement on créera un constructeur :

```
public ClientSalarie(Salarie salarie, ....) {
          super(...) ;
          this.sal = salarie ;
          ....
}
```

Introduction: un cas particulier d'héritage/délégation



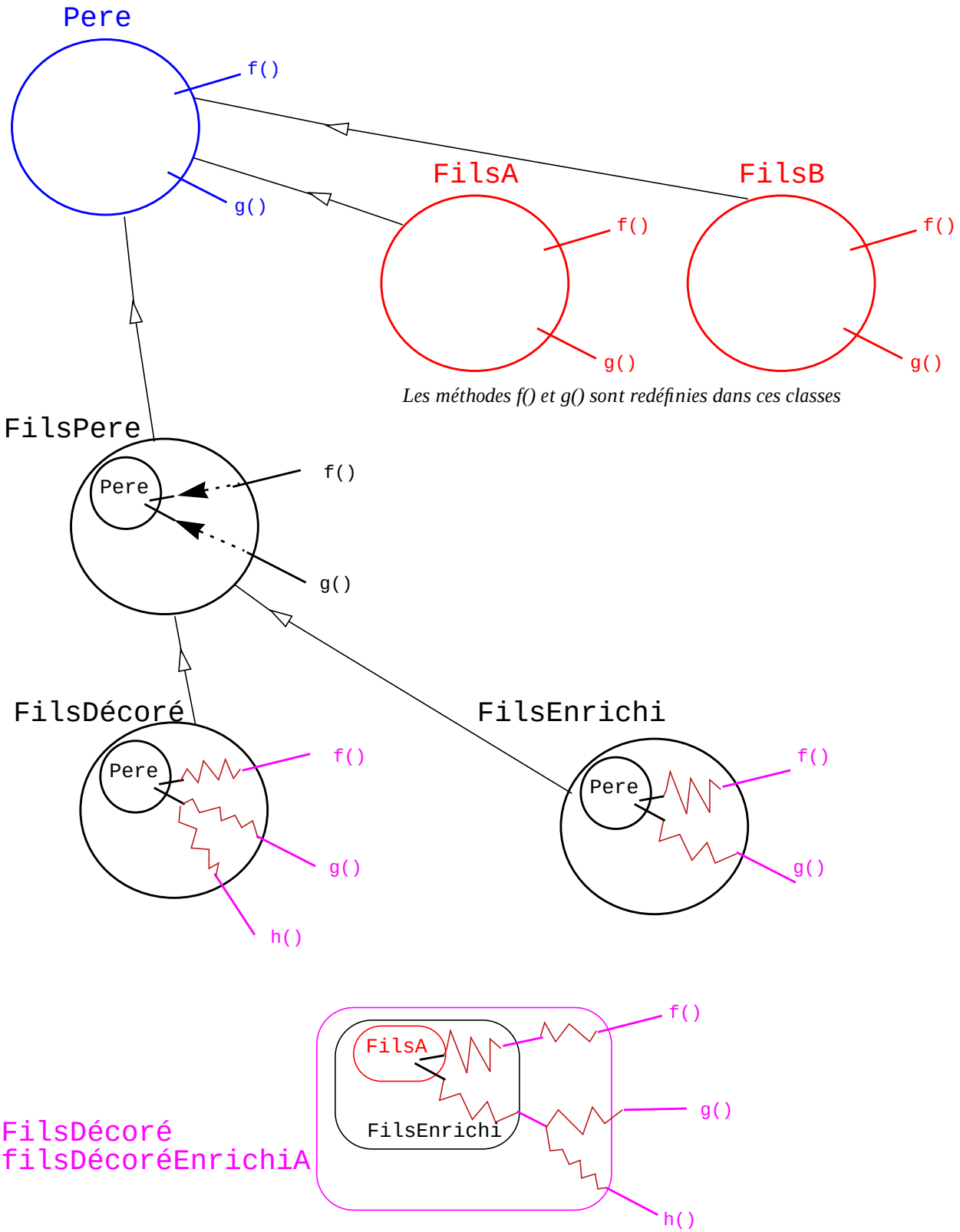
Ici chaque méthode de la super-classe est redéfinie pour être déléguée ... à une instance de la super classe!

Tous les constructeurs prennent en paramètre une instance de la super-classe!

```
class FilsPere extends Pere {
        private Pere pere ;

        public FilsPere(Pere pere) {
                this.pere = pere ;
        }

        public void f() {
                pere.f() ;
        }
        ...
}
```

A quoi peut bien servir un dispositif aussi étrange?

Un arbre d'héritage complexe...



Un arbre d'héritage complexe

et même très complexe!:

- Les classes `FilsA` et `FilsB` héritent de `Père` en spécialisant ses méthodes. Le comportement de ces méthodes est donc différent et spécifique aux classes `FilsA` et `FilsB`.

- La classe `FilsPere` correspond au dispositif vu précédemment, n'oublions pas que son constructeur est `FilsPere(Père)`

- Les classes `FilsEnrichi` et `FilsDécoré` héritent de `FilsPere`. Leurs méthodes peuvent être redéfinies mais s'appuient *in fine* sur les méthodes de même nom de la classe `Père`. Dans le cas de FilsEnrichi on créée de nouvelles méthodes dont la réalisation s'appuie sur les méthodes fondamentales de la classe `Père`.
  Les constructeurs de ces classes utilisent toujours une instance de `Père`.

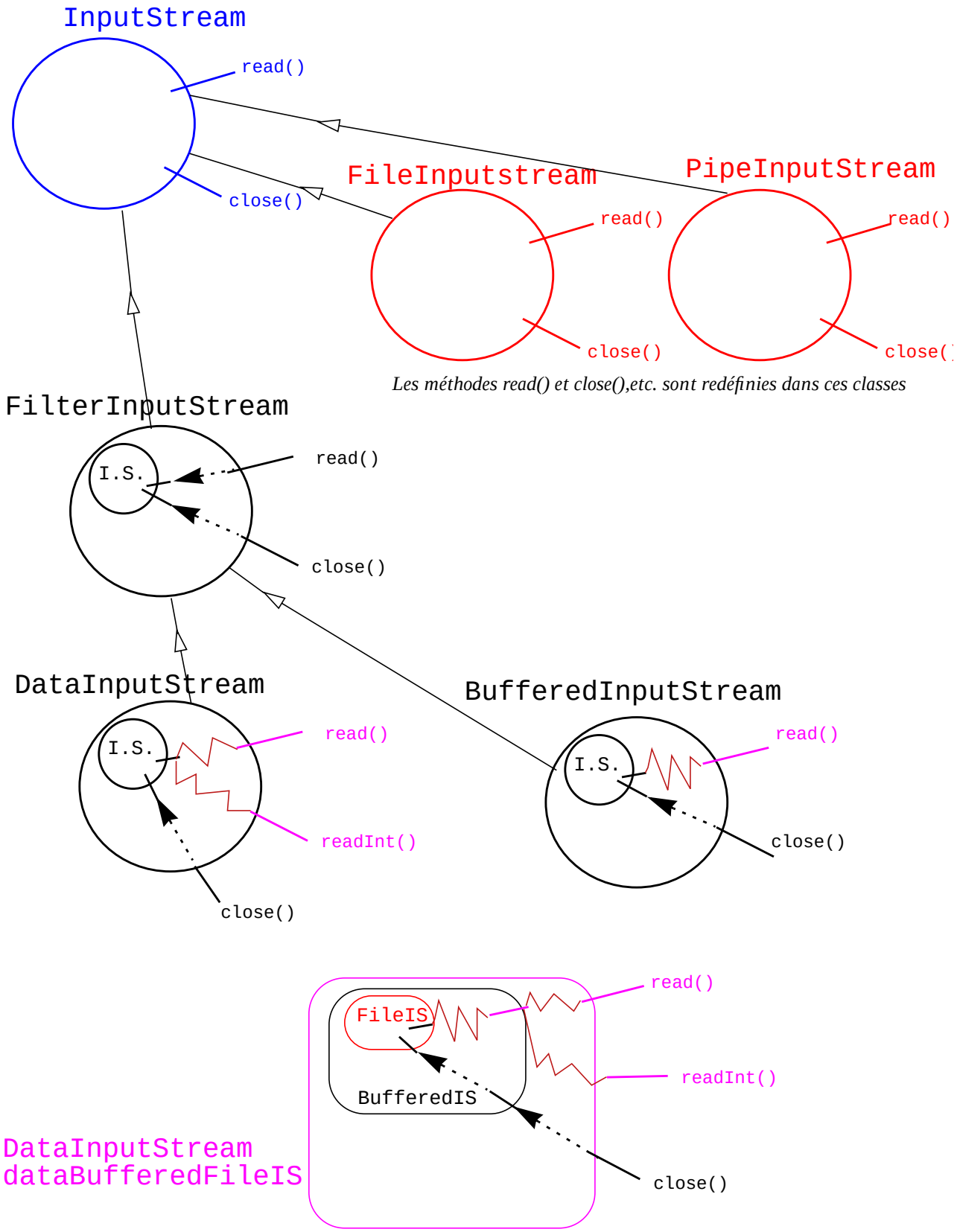
- On peut alors construire une instance de `FilsDécoré` qui s'appuie sur les comportements de `FilsEnrichi` **et** de `FilsA`:

```
new FilsDecore(new FilsEnrichi(new FilsA())) ;
```

On a ainsi composé des comportements en agissant de manière transversale dans l'arbre d'héritage.

Nous allons donner un exemple pratique de ce modèle structurel.

Application du modèle structurel aux E/S "flot" en Java:



Application du modèle structurel aux E/S "flot" en Java:

Les flots (*streams*) d'E/S sont basés sur un concept simple et unificateur. Des octets "coulent" dans un flot et au point d'accès du dispositif on consomme des octets les uns après les autres (`InputStream`) ou on fourni des octets les uns après les autres (`OutputStream`).

Le principe d'utilisation est de programmer en s'adressant à un point d'accès qui délégue la réalisation de l'entrée/sortie à une combinaison de dipositifs. Pour réaliser ces combinaisons on dispose:

- d'une classe au sommet de la hiérarchie. Ici il s'agit de `InputStream` (qui est d'ailleurs une classe abstraite).
- de classes réalisant des E/S fondamentales: les ressources (*node streams*). Ainsi ces classes permettent l'ouverture d'`InputStream` sur des fichiers, des sockets, des tableaux en mémoire, des "pipes" (échanges entre *threads* à l'intérieur d'un processus JVM), etc. Ces classes héritent de `InputStream` et implantent les comportements fondamentaux qui sont la base du service.
- d'une classe `FilterInputStream` dont l'objectif est purement structurel (on notera que son constructeur est `protected`).
- de classes "filtres" qui transforment les comportements de flots existant: `BufferedInputStream` (bufferisation utile pour les performances), `DataInputstream` (permet de lire des primitifs Java sur un flot), etc.

On peut alors réaliser des instances ayant des comportement spécifiques:

```
new DataInputstream(new BufferedInputStream(
        new FileInputStream("fichier"))) ;
```

Si besoin est, on pourrait lire des objets Java de manière cryptée, compressée, bufferisée au travers d'une *socket* réseau ! (voir les packages `java.util.zip`, `java.security` et `javax.crypto`)

Flots d'octets (InputStream, OutputStream)

La définition fondamentale des flots est de décrire un dispositif pour lire / écrire des octets (`byte`).

Ces classes sont donc munies de méthodes de bas niveau comme :

```
java.io.InputStream
         int read() throws IOException
          // lit en fait un byte dans un int
          // + d'autres read(...) de bas niveau

java.io.OutputStream
         void write(int octet) throws IOException
          // écriture d'un byte passé comme int
          // + autres write(...) de bas niveau
```

Les deux classes sont munies d'une méthode de fermeture

```
         void close() throws IOException
```

Flots de caractères (Reader, Writer)

Dans la mesure où Java utilise de manière interne des caractères UNICODE codés sur 16 bits on a été amené à définir d'autres types fondamentaux de flots adaptés aux caractères. Ce sont les **`Reader`**s et les **`Writer`**s.

```
java.io.Reader
         int read() throws IOException
          // lit en fait un char dans un int
          // + d'autres read(...) de bas niveau

java.io.Writer
         void write(int carac) throws IOException
          // écriture d'un caractère passé comme int
          // + autres write(...) de bas niveau
          // dont write(string s)
```

Bien entendu ces classes disposent également de `close()`.

Typologie par “ressources”

En prenant l’exemple des seuls flots d’entrées comparons les `InputStream`s et les `Reader`s

catégorie	**`InputStream`**	**`Reader`**
lecture dans tableau mémoire	`ByteArrayInputStream`	`CharArrayReader` `StringReader`
mécanisme producteur/consommateur entre Threads	PipedInputStream	PipedReader
fichiers	FileInputStream	FileReader **InputStreamReader**
flots fabriqués par d’autres classes	java.lang.Process getInputStream() java.net.Socket getInputStream() java.net.URL openStream() ...	

Une classe permet de passer du domaine des `InputStream`s à celui des `Reader`s : **`InputStreamReader`**

Conversions octets-caractères

On utilise les caractères UNICODE à “l’intérieur” du monde Java, mais de nombreuses ressources texte n’utilisent pas ce mode de codage. On a ainsi des fichiers “texte” qui peuvent avoir été fabriqués par des systèmes divers qui emploient des jeux de caractères différents (le plus souvent avec des caractères codés sur 8 bits).

Lorsqu’on échange du texte entre le “monde” Java et le monde extérieur il est essentiel de savoir quel est le mode de codage de texte adapté à la plateforme cible.

Deux classes `InputStreamReader` et `OutputStreamWriter` permettent de passer d’un flot d’octets à un flot de caractères on opérant les conversions appropriées.

```
try {
      InputStream fis = new FileInputStream("fichier.txt") ;
      Reader ir = new InputStreamReader(fis,"ISO-8859-1") ;
         .....
} catch (UnsupportedEncodingException exc) {....// log
} catch (FileNotFoundException exc) {....// log
}
```

Le second argument du constructeur indique le mode de codage (“ISO-8859-1” ou “ISO-8859-15” dans la plupart des pays européens, “Cp1252” sous Windows). La liste des codages acceptés se trouve dans la documentation sous `docs/guide/internat/encoding.doc.html` - voir aussi l’outil `native2ascii` ou la documentation 1.4 de `java.nio.charset`- On notera qu’il existe un codage nommé UTF8 qui permet de fabriquer un flot de caractères UNICODE codés sur 8 bits.

Il existe également des constructeurs pour ces deux classes dans lesquels on ne précise pas le mode de codage: c’est le mode de codage de la plate-forme locale qui est pris par défaut.

A partir de la version 1.4 on peut utiliser également un argument de la classe `Charset` (permet d’accéder à un meilleur contrôle sur le mécanisme de conversion).

Filtres

Les *filtres* sont des dispositifs qui transforment la manière dont un flot opère: bufferisation, conversion des octets en données Java, compressions, etc.

- Les filtres sont des flots dont les constructeurs utilisent toujours un autre flot:

```
// exemple de déclarations des constructeurs
public BufferedInputStream(InputStream in) ...
public DataInputStream(InputStream in)...
```

- On utilise les filtres en les associant avec un flot passé en paramètre du constructeur. Ce flot peut lui même être un autre filtre, on combine alors les comportements:

```
// dans des blocs try/catch ....
BufferedInputStream bis = new BufferedInputStream(
                  new FileInputStream (nomFichier)) ;

DataOutputStream dos = new DataOuputStream(
                  new BufferedOutputStream (
                            new FileOutputStream (nomFic))) ;
```

- Chaque filtre dispose de méthodes spécifiques, certaines méthodes ont la propriété de remonter la chaîne des flots :

```
try {
         dos.writeInt(235);
         dos.writeDouble(Math.PI);
         dos.writeUTF(”une chaîne accentuée”) ;
                  // chaîne écrite en UTF8
         dos.flush() ; //transmis au BufferedOutputStream
         ...
} catch (IOException exc) {
         /*log */
} finally {
         try {
                  dos.close() ; // ferme toute la chaîne
         } catch (IOException exc) { /*ignore */}
}
```

Filtres courants

InputStream/OutputStream

- `Buffered`* : la bufferisation permet de réaliser des E/S plus efficaces (en particulier sur le réseau).
- `Data`* : permet de lire/écrire des données primitives Java
- `Object`* : permet de lire/écrire des instances d'objets. Ce type de flot est très important (Voir *Dispositifs avancés d'E/S*, page 197.).
- `PushBackInputStream`: permet de "remettre" dans le flot des données déjà lues.
- `SequenceInputStream`: permet de considérer une liste ordonnée de flots comme un seul flot.

Reader/Writer

- `Buffered`* : bufferisation pour des caractères. Permet de lire un texte ligne à ligne.
- LineNumberReader : BufferedReader avec numérotation des lignes.
- `PrintWriter`: pour disposer des méthodes print/println (voir aussi `PrintStream`)
- `StreamTokenizer` : permet de faire de l'analyse syntaxique de texte.

Dans d'autre packages on trouvera des filtres spécialisés : java.util.zip.ZipInputStream, java.security.DigestInputStream,...

Il existe également des dispositifs d'entrée/sortie qui ne s'inscrivent pas dans la logique des Streams : `RandomAccessFile` (fichier à accès direct) et mécanismes de `java.nio` en java 1.4.

Exercice: pratique des E/S flot

Exercice :

- Reprendre l'interface graphique Question-Réponse et lui associer le comportement suivant :
  - l'utilisateur saisit une chaîne de caractère
  - si cette chaine est une URL (voir constructeur `java.net.URL`) ouvrir un `Stream` avec la méthode `openStream()` et si cette ouverture réussit lire le contenu et l'afficher
  - sinon on considérera que la chaîne saisie est celle d'un fichier texte local, le lire et l'afficher

  Pour réaliser correctement l'exercice nous vous conseillons les étapes suivantes:
  - Rechercher dans la documentation la méthode qui permet de lire un texte ligne à ligne.
  - Tenter d'ouvrir un fichier texte et lire la première ligne
  - Lire toutes les lignes
  - Réaliser l'ensemble de l'exercice.
- Suite de l'exercice précédent: comment modifier le principe général de l'interface graphique pour la faire collaborer plus efficacement avec des codes du type de celui que vous venez d'écrire?

Compléments

Les compléments techniques suivants constituent une annexe de référence

RandomAccessFile

Exemple d'ouverture :

```
RandomAcessFile raf ;
try {
        raf = new RandomAccessFile("stock.raf","rw") ;
catch(Exception exc) {
        ....
}
```

Le premier argument donne le nom du fichier impliqué, le second indique le mode d'accès ("r" pour lecture seule, "rw" pour lecture/écriture -dans ce cas si le fichier n'existe pas il est créé-).

Les exceptions susceptibles d'être levées concernent le mode (chaîne incorrecte), la désignation ou les droits d'accès au fichier .

La lecture et écriture de données peut s'opérer sur des données primitives scalaires (int, byte, double, char,...) et sur des chaînes de caractères (capacités analogues aux `DataStream`) -toutefois dans ce dernier cas il devient difficile de contrôler correctement les accès directs en fonction d'une taille prédéterminée-.

Pour gérer l'accès direct :

```
void seek (long position) ; //positionner dans le fichier
                    // commence en origine zero

long getFilePointer() ; // position actuelle

long length() ; // position fin de fichier
```

Les “design patterns”

Les “*design patterns*” sont un sujet à la mode et il existe une abondante littérature sur le sujet (le livre de référence est celui de Gamma, Helm, Johnson, Vlissides).

Le problème principal de leur pratique est de comprendre la portée d’un *pattern* et de savoir l’adapter à un problème particulier. Pour s’en convaincre rechercher une description du modèle “décorateur” et le comparer à l’utilisation qui en est faite dans les E/S Java. On peut aussi analyser les limites du modèle ou de son implantation (pour s’en convaincre regarder la finalité et la position dans le schéma de la méthode `flush()`).

Programmation réseau 10

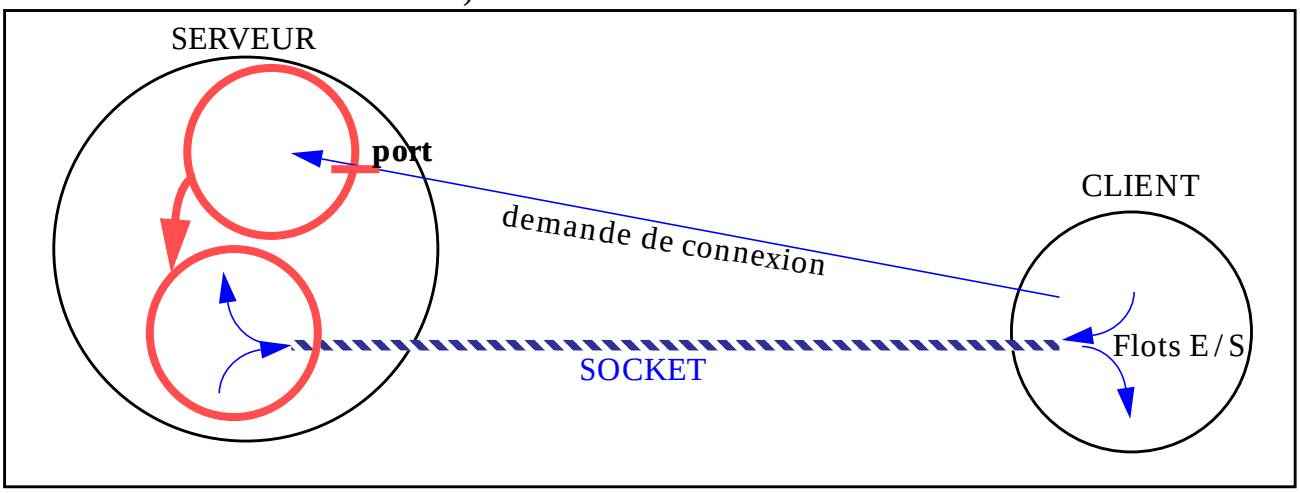
Points essentiels

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.

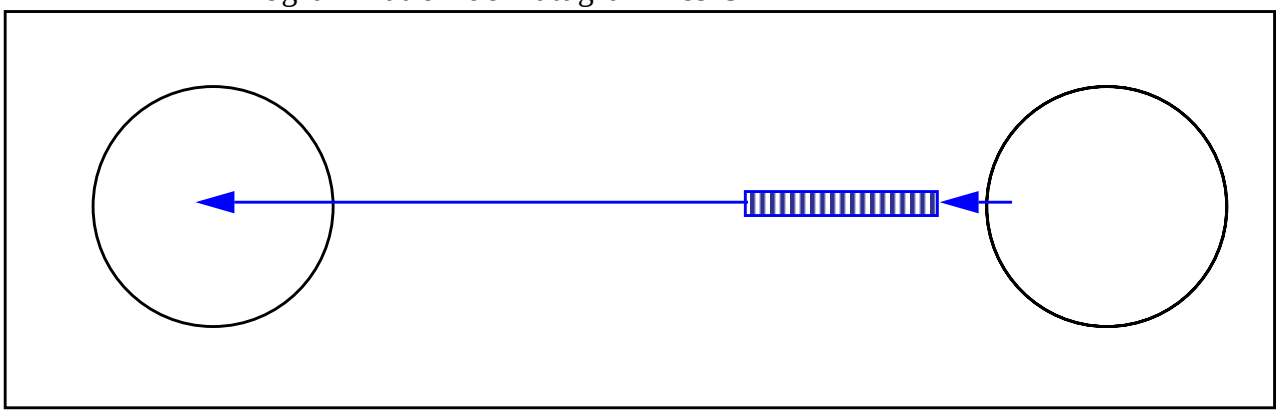
- communications sur TCP (flots)
- communications via UDP (datagrammes).
- datagrammes avec diffusion multiple (multicast).

Modèles de connexions réseau en Java

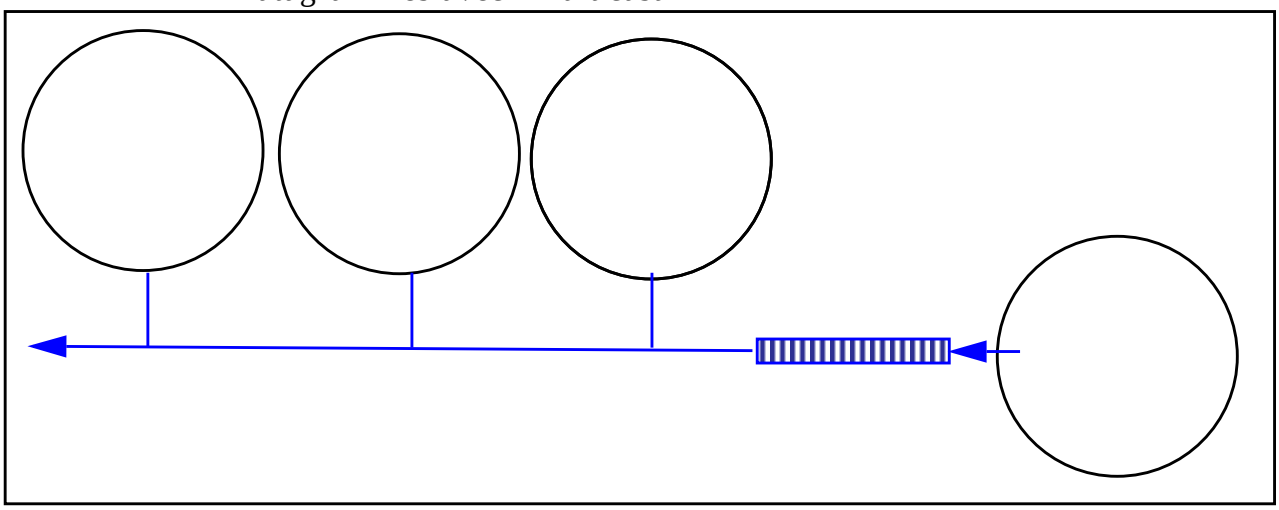
Accès programmatiques aux “sockets” (points d’accès aux interfaces de communication) sur TCP/IP



Programmation de Datagrammes UDP



Datagrammes avec “Multicast”



Sockets

Les `Socket`[1]`s` 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.

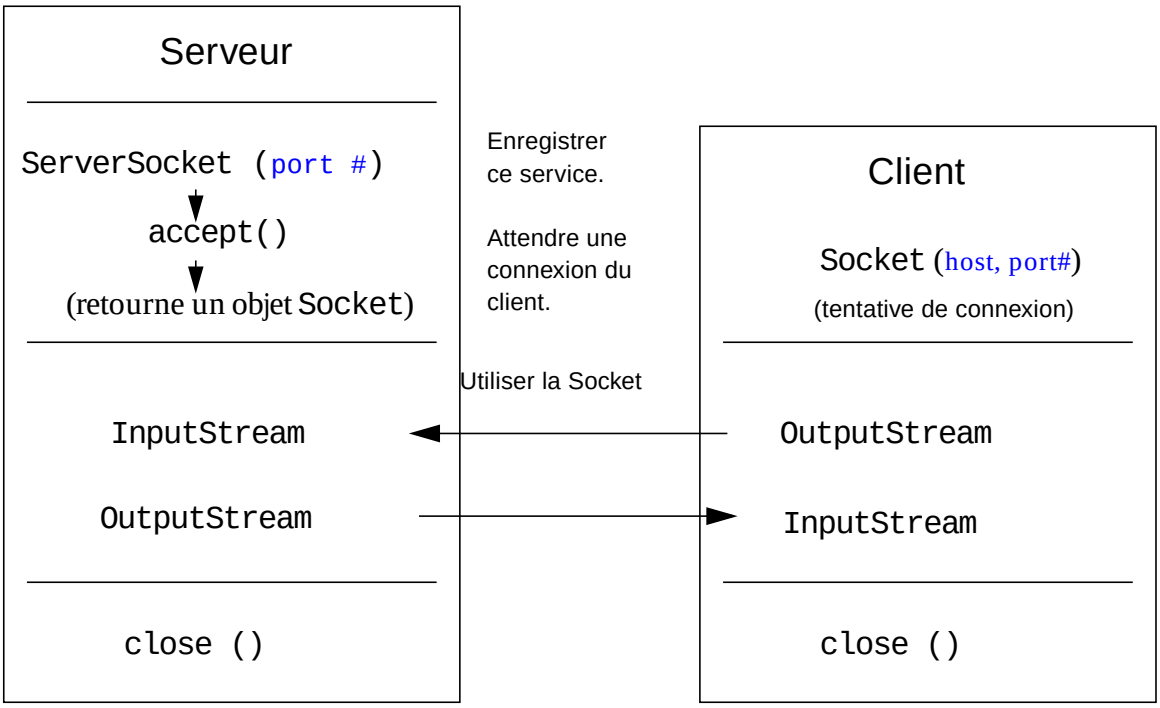
- **Datagram sockets (UDP)** – Permettent d'établir une connexion en mode non-connecté, que l'on appelle aussi mode Datagramme. Les données doivent être assemblées et envoyées sous la forme de paquets indépendants de toute connexion. Un service non connecté est généralement plus rapide qu'un service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

1. En fait une `Socket` est une abstraction de plus haut niveau qui permet des réalisations très diverses. On associe: *interface réseau* (accès à une liaison physique), adresse réseau, type de protocole et numero de port.

Sockets TCP/IP.

Dans Java, les sockets TCP/IP sont implantées au travers de classes du package `java.net`.

Diagramme d'utilisation:



Dans ce modèle, le fonctionnement est le suivant :

- Le serveur enregistre son service sous un numéro de port. 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.

Serveur TCP/IP (exemple simple)

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

public  class Serveur {
    private ServerSocket srv ;

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

   public void go() {
      while(true) {
         try {
            Socket sck = srv.accept() ; //bloquant
            OutputStream os = sck.getOutputStream() ;
            DataOutputStream dos = new DataOutputStream(os) ;
            dos.writeUTF("Hello Net World!") ;
            sck.close() ;
         } catch(IOException exc) {
            Logger.global.log(Level.SEVERE,"serveur",exc) ;
         }
      }
   }

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

Dans cet exemple l'échange client/serveur est extrèmement simple : à la connexion c'est le serveur qui émet et qui prend ensuite l'initiative de couper la connexion.

Bien entendu d'autres possibilités sont ouvertes:

- Le client pourrait être le premier à émettre (et donc le serveur devrait commencer par "lire")
- La connexion pourrait rester ouverte pour un succession d'échanges entre client et serveur.

Client TCP/IP (exemple simple)

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

public  class Client {
    private Socket sck ;

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

   public void go() {
         try {
            DataInputStream dis =
                        new DataInputStream(sck.getInputStream()) ;
            String msg = dis.readUTF() ;
            Logger.global.info("reçu : " +msg) ;
         } catch(IOException exc) {
            Logger.global.log(Level.SEVERE,"serveur",exc) ;
         }
   }

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

Ici le client ne se pose aucune question sur la durée de la communication puisque c'est le serveur qui prend l'initiative de couper la connexion.

On notera dans les deux exemples que les constructeurs sont susceptible d'échouer (numero de port dejà pris du coté du serveur, problème de désignation de l'hôte du serveur, accès à cet hôte, absence de processus serveur, etc. du coté du client).

Serveur TCP/IP (exemple avec threads)

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

public abstract class ServeurThread {
    protected ServerSocket srv ;

   protected ServeurThread(int port) throws IOException{
      srv = new ServerSocket(port) ;
   }

   public void go() throws IOException{
      ThreadGroup thg = new ThreadGroup("serveur"){
          public void uncaughtException(Thread t, Throwable e) {
            Logger.global.log(Level.SEVERE,t.toString(), e) ;
         }
      } ;
      while (true) {// mettre un système d'arrêt propre!
         Socket sck = srv.accept() ;
         new Thread(thg, fabriqueSession(sck)).start() ;
         // créer un Pool de Threads réutilisables ne serait pas mal
      }
   }

   public abstract Runnable fabriqueSession(Socket sck) ;
}
```

Exemple de définition de la méthode “fabrique” dans une sous-classe:

```
public  Runnable fabriqueSession(final Socket sck)  {
      return new Runnable () {
         public void run() {
            try {
               DataInputStream dis =
                       new DataInputStream(sck.getInputStream()) ;
               ....// codes
               sck.close() ;
            } catch (IOException exc) {
               Logger.global.log(Level.SEVERE,"thread serv",exc) ;
            }
         }// run
      } ; // Runnable
   }
```

Point intermédiaire : échanges TCP

Mini-exercice:

- Chaque stagiaire doit écrire un client et un serveur utilisant les communications sur Socket TCP:
  - Chaque serveur est un programme autonome qui écoute sur le port 6666 et qui envoie un message secret à tout client qui se connecte. Ce message secret est un tableau de `String` local au serveur. Les envois se feront par `writeUTF()` (de `DataOutputStream`).
  - Chaque client est un code lié à l'interface graphique : lorsque l'utilisateur tape le nom d'un hôte sur le réseau, le code contacte le serveur correspondant et affiche le message secret.

Echanges U.D.P.

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 point d'accès (socket) UDP. Pour un serveur on précisera le port (pas nécessaire pour le client initiateur de l'échange)

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

- `InetAddress` : permet de produire une adresse inet à partir d'une désignation (méthode `getByName`()) ou de la machine locale (`getLocalHost`())
  L'adressage d'un datagramme nécessite de préciser une adresse Inet + un numero de port cible. On peut aussi utiliser une classe plus synthétique: `InetSocketAddress` (sous-classe de `SocketAddress`)

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

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

Serveur U.D.P. (exemple)

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

public  abstract class ServeurUDP {
   public static final int DATA_MAX_SIZE = 512 ;
    protected DatagramSocket veille ;

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

   public void go() {
      byte[] recvBuff = new byte[DATA_MAX_SIZE] ;
      while(true) { // prevoir arrêt!
         try {
            /* on écoute */
            DatagramPacket recvPack =
                   new DatagramPacket(recvBuff, recvBuff.length) ;
            veille.receive(recvPack) ; // bloquant
            /* reçu */
            byte[] reponse =  reponseA(recvPack.getData()) ;
            /* on renvoie reponse  a l'expéditeur*/
            DatagramPacket sendPack =
               new DatagramPacket(reponse, reponse.length,
                     recvPack.getSocketAddress()) ;
            veille.send(sendPack) ;
         } catch(IOException exc) {
            Logger.global.log(Level.SEVERE,"serveur udp",exc) ;
         }
      }
   }

   public abstract byte[] reponseA(byte[] question) ;
}
```

Les données échangées sont ici stockées dans un tableau de `byte`. Si la structure interne est un peu compliquée on peut utiliser des `ByteArrayInputStream` ou des `ByteArrayOutputStream` pour stocker/lire des données (voir aussi `java.nio` en 1.4).

Client U.D.P. (exemple)

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

public  class ClientUDP {
    public static final int DATA_MAX_SIZE = 512 ;
    protected DatagramSocket dsk  ;
    protected InetSocketAddress cible ;

   public ClientUDP(String host, int port)throws IOException {
      dsk = new DatagramSocket();
      cible = new InetSocketAddress (
                  InetAddress.getByName(host), port) ;
   }

   byte[] questionReponse(byte[] données) throws IOException {
      DatagramPacket sendPack =
         new DatagramPacket(données, données.length, cible) ;
      dsk.send(sendPack) ;
      DatagramPacket recvPack =
         new DatagramPacket(new byte[DATA_MAX_SIZE],
                                    DATA_MAX_SIZE) ;
      dsk.receive(recvPack) ; //bloquant
      return recvPack.getData() ;
   }
}
```

On notera qu'ici la constructeur de `DatagramSocket` peut choisir lui-même son numéro de port. L'information est contenue dans le datagramme et peut-être exploitée par le serveur pour renvoyer un datagramme à la bonne adresse. Comme dans tous ces exemples on notera que les reférences à des `SocketAddress` n'existent pas dans des versions antérieures à la 1.4 .

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 `TimeToLive` 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)

Diffuseur Multicast (exemple)

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

public  class Radio  {
    public static final int DATA_MAX_SIZE = 512 ;
    public static final int PORT = 6969 ;
    public static final String GROUP = "229.69.69.69" ;
    public static final int TTL = 1 ;


   public static void main (String[] args) throws Exception {
      int max = Integer.parseInt(args[0]) ;
      MulticastSocket emetteur = new MulticastSocket();
                                        //pas de port!
      emetteur.setTimeToLive(TTL) ;
      InetSocketAddress canal = new InetSocketAddress(
               InetAddress.getByName(GROUP), PORT) ;
      for(int ix = 0 ; ix < max ; ix++) {
         byte[] contenu = { (byte) ix} ;
         DatagramPacket sendPack =
                   new DatagramPacket( contenu,1,  canal) ;
         emetteur.send(sendPack) ;
      }
   }
}
```

Ecouteur Multicast (exemple)

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

public  class Ecouteur  {

   public static void main (String[] args) {
      int max = Integer.parseInt(args[0]) ;
      MulticastSocket recepteur =null ;
      InetAddress groupe =  null ;
      try {
         recepteur = new MulticastSocket(Radio.PORT); //  port!
         byte[] vals = new byte[Radio.DATA_MAX_SIZE] ;
         groupe = InetAddress.getByName(Radio.GROUP) ;
         recepteur.joinGroup(groupe) ;
         DatagramPacket recvPack =
            new DatagramPacket(vals, vals.length) ;
         for(int ix = 0; ix <max; ix++) {
            recepteur.receive(recvPack) ;
            Logger.global.info(Byte.toString(vals[0])) ;
         } //for
      } catch(IOException exc) {
         Logger.global.log(Level.SEVERE,"multicast", exc) ;
      } finally {
         if(null != recepteur && null != groupe) {
            try {recepteur.leaveGroup(groupe) ;
            }catch(IOException exc) {/*ignore*/}
         }
      }

   }
}
```

Point intermédiaire : échanges UDP-Multicast

Exercice (à faire chez soi : l'exercice est long et cette partie du cours est facultative)

- Pour les amateurs de Loto traditionnel (appelé aussi Raffle dans certaines régions) réaliser un jeu de loto en réseau:
  - pour chaque poste de joueur on génère une "carte" de loto (3 rangées de 9 cases), seules 15 cases portent un chiffre.
  - le "maitre de jeu" (un serveur multicast) publie à intervalles réguliers un numéro au hasard compris entre 1 et 90.
  - le joueur qui gagne est celui qui a coché toutes ses cases et qui réagit le plus vite (et qui ne triche pas!).

  Voir class `java.util.Random` pour générer des nombres aléatoires.

Dispositifs avancés d'E/S 11

Points essentiels

Deux aspects très différents des systèmes d'entrée/sortie disponibles en Java:

- Un dispositif de la famille des flots (*streams*) les entrées/sorties "objet": `ObjectInputStream`, `ObjectOutputStream`
- Une catégorie de système d'entrée/sortie ne suivant pas le modèle des flots: la package `java.nio`

E/S d'objets

Les classes `DataInputStream`, `DataOutputStream` pemettent de lire/écrire des données java dans un flot: données scalaires (`readBoolean`, `readInt`, `readDouble`, etc.) et chaîne de caractères UNICODE codées en UTF-8 (`readUTF`).

Les classes `ObjectInputStream` et `ObjectOutputStream` disposent des mêmes fonctionnalité mais permettent en plus de lire ou écrire des instances d'objet :

```
try {
        ObjectOutputStream oos =
             new ObjectOutputStream(
                  new BufferedOutputStream(
                       new FileOutputStream(nomFichier)));
        oos.writeObject(new BigDecimal("6.55957")) ;
        oos.flush() ;
        ...
} catch(IOException exc) { /*log correct */}
```

Et inversement :

```
try {
        ObjectInputStream ois =
             new ObjectInputStream (
                  new BufferedInputStream(
                       new FileInputStream(nomFichier))) ;
        BigDecimal taux = (BigDecimal) ois.readObject() ;
        ....//
}catch(ClassNotFoundException xc) { /* log */
}catch(IOException exc) { /* log*/
}
```

On notera que `readObject()` étant déclarée avec un résultat de type `Object` on doit utiliser une opération de projection de type (*cast*) pour pouvoir écrire une affectation correcte. D'autre part il n'est pas sûr que la JVM dispose du type effectif de l'objet transmis par le flot d'où la nécessité de capter l'exception correspondante (`ClassNotFound`); par ailleurs la lecture d'un objet dans le flot alors que c'est un primitif qui s'y trouve provoquera un erreur d'entrée/sortie.

Les objets dans un flot

Tous les objets sont-ils susceptibles d'être envoyés dans un flot (fichier, ligne de communication, etc.)?

Et d'abord quelles sont les données qui sont concernées par le processus de linéarisation de l'instance (*serialization*). Ici l'objectif est de transmettre **l'état** de l'instance pour pouvoir la reconsituer, outre le fait que l'émetteur et le récepteur de l'objet doivent avoir un minimum d'accord sur la classe correspondante, il faut comprendre que:

- Certains objets sont, par nature, non susceptibles d'être transmis parce que leur état réel n'est pas reproductible, et/ou parce que ces instances dépendent d'éléments de contexte qui ne sont pas transmissibles entre JVM. Citons, par exemple, les *threads* ou les dispositifs d'entrée-sortie eux-mêmes.
  Les dispositifs d'entrée/sortie d'objets ont donc besoin d'une information leur indiquant explicitement que la linéarisation est licite. Les objets concernés doivent donc implanter une des interfaces **`Serializable`** (protocole controlé par le dispositif d'E/S) ou **`Externalizable`** (protocole spécifique défini et controlé par le programmeur).
  Une tentative d'écriture d'un objet non-`Serializable` provoque une `NotSerializableException`.

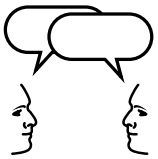
- Il peut être souhaitable de soustraire certaines variables membres de l'instance au processus de linéarisation (par exemple parce que ces variables ne sont pas `Serializable`!).
  Un modificateur particulier: **`transient`** permet d'indiquer que le champ n'est pas transmissible.

- Les variables partagées (membres `static`) ne sont pas transférées.
  Les autre modificateurs comme `private`, `protected`, etc. n'ont aucun effet sur le mécanisme de linéarisation.

Objet adapté à la linéarisation (exemple)

En supposant une classe `Adresse` marquée `Serializable`:

```
import java.io.* ;

public class Salarie implements Serializable {
   public final String id ;
   private String nom ;
   private Adresse adresse ;
   private transient Salarie manager ;
   ....

   public Salarie(String nom, String id) {
      this.id = id ;
      this.nom = nom ;
      .....
   }

   public void associeManager(String managerId) {
         ...//
   }
   ...
}
```

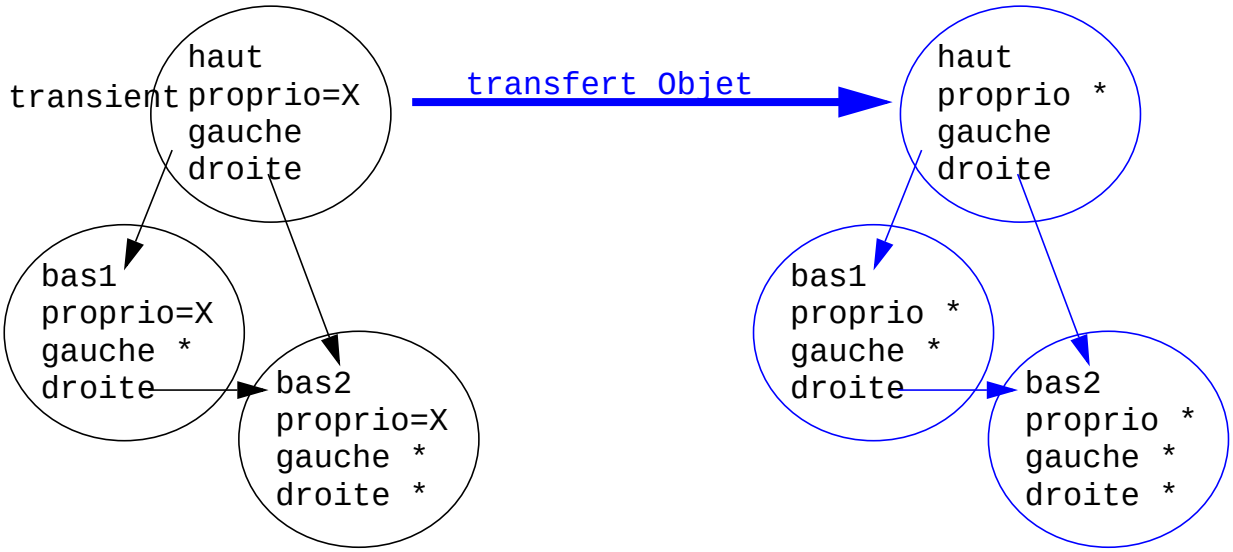


Discussion: Quelle raisons peut-on avoir pour associer le modificateur `transient` *au champ “manager”?*

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 le flot mémorise les instances. Pour que le Stream "oublie" les instances dejà transférées utiliser la méthode `reset()`.

- La création d'un `ObjectInputStream` est un **appel bloquant**. On ne sortira de cet appel que lorsque le flot aura reçu des informations qui lui permettent de s'assurer que le protocole avec l'`ObjectOutputStream` correspondant est correct. Ces informations se trouvent dans les données initiales du flot (écrites par l'ouverture de `ObjectOutputStream` correspondant).

- L'utilisation la plus simple du mécanisme de linéariisation 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

Modifications des opérations liées aux objets Serializable

On peut personnaliser de diverses manières la façon dont la linéarisation opère en traitant une instance. Le mode opératoire principal consiste à déclarer dans la classe deux méthodes symétriques de responsabilité `private`.

Reprenons l'exemple précédent de la classe `Salarie`:

```
public class Salarie implements Serializable {
   public final String id ;
   private String nom ;
   private Adresse adresse ;
   private transient Salarie manager ;
   .... ;
   private void writeObject(ObjectOutputStream oos)
                                        throws IOException {
         oos.defaultwriteObject() ;
         if( null != manager ) {
                  oos.writeUTF(manager.id) ;
         } else {oos.writeUTF("") ;}
   }

   private void readObject(ObjectInputStream ois)
                                        throws IOException {
         ois.defaultReadObject() ;
         String idManager = ois.readUTF() ;
         associeManager(idManager) ;
   }
```

Par convention ces deux méthodes `writeObject`, `readObject` sont prises en compte par le mécanisme de linéarisation.

Ici les méthodes de `ObjectStream defaultRead/Write` permettent d'opérer la lecture/écriture "normale" de l'instance, mais ensuite on complète le protocole avec d'autres données.

Point intermédiaire : échanges d'objets

Mini-exercice :

- Toujours un client graphique et un serveur autonome :
  - Le client qui veut se connecter au serveur, génère un objet de type `java.sql.Timestamp` par :

```
new Timestamp(System.currentTimeMillis()) ;
```

  - L'objet est envoyé au serveur qui le lit et le renvoie accompagné d'un autre `Timestamp` généré sur le serveur.
  - Le code client lit les deux `Timestamp`, en génère un troisième et affiche les 3 objets. (on pourra ainsi constater les dérives éventuelles des horloges!).
- Pour les très rapides: sur le serveur tracez les caractéristiques de votre interlocuteur (adresse IP).

Le package `java.nio`

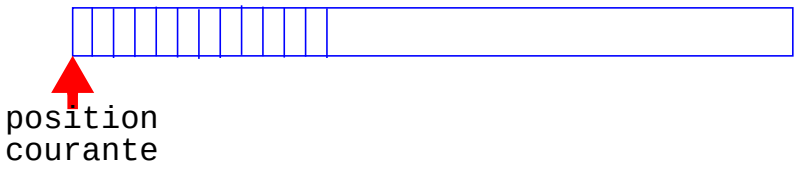
Le package `java.nio` introduit un certain nombre de mécanismes d'entrée/sortie qui ne se situent pas dans la philosophie des flots:

- Les entrées/sorties sont opérées indirectement sur des `Buffer`s.
- Le type principal de `Buffer` est `ByteBuffer`, mais il existe aussi des `CharBuffer`, `IntBuffer`, etc. Chacun a des opérations spécifiques pour lire ou écrire différents types de données. Lectures et écritures peuvent éventuellement se produire sur le même `Buffer` et il y a une "position" courante qui se déplace en conséquence.
- Le package `java.nio.channels` définit des *channel*s c.a.d des dispositifs d'entrée/sortie. Une série d'interfaces définissent par exemple des `Channel`s capable de lire ou d'écrire à partir de `ByteBuffer`s. Le point important est que ces opérations peuvent être asynchrones (mais atomiques du point de vue des *thread*s: un *thread* qui tente une opération d'E/S sur un *channel* bloque tant qu'une autre opération d'E/S est en cours sur ce même *channel* et n'est pas terminée -ou interrompue-).
- Chaque type effectif de `Channel` peut avoir des caractéristiques propres susceptibles d'être exploitées.
  Les `FileChannel`s permettent, par exemple, d'avoir un "mapping" direct entre le fichier système et un `ByteBuffer` correspondant (technique performante uniquement sur de très gros fichiers); ils permettent également de mettre des verrous sur des fichiers ou des portions de fichiers (toutefois il n'est pas garanti que ces verrous soient opposables à d'autre processus que la JVM courante -ceci est dépendant du système-).
  Les `SocketChannel`s et `ServerSocketChannel`s sont des exemples de channel *selectable*. Un seul *thread* peut être bloqué sur une opération `select`, au "réveil" il pourra obtenir la liste des *channel*s sur lesquels une opération d'entrée/sortie est demandée. Ce dispositif permet de réaliser des serveurs performants qui n'ont pas besoin de faire "vivre" autant de *threads* qu'il y a de *sockets* disponibles.

Mise en oeuvre de Buffers

1. Allocation d'un Buffer

```
int taille = 2 + 30 ;
ByteBuffer buff= ByteBuffer.allocate( taille ) ;
```



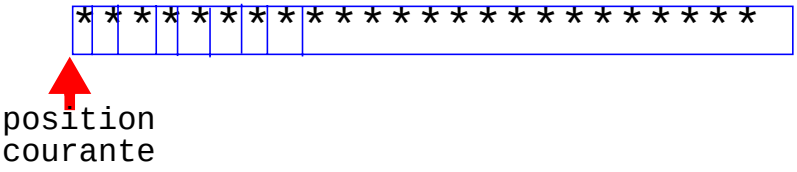
Il est également possible d'avoir une allocation avec corrrespondance directe avec un dispositif sous-jacent (*mapping* avec dispositif de lecture / écriture d'un fichier par ex.). Dans ce cas les `read`s explicites et les `rewind`s des phases 2 et 3 ne sont pas nécessaires.

2. Lecture (par exemple) à partir d'un Channel :

```
ReadableByteChannel chan =
         new RandomAccessFile(nomFic,"rw").getChannel() ;
chan.read(buff) ;
```

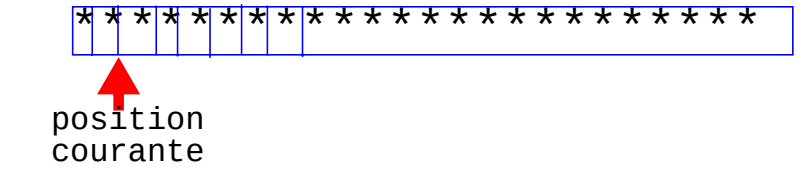
3. Pour exploitation du Buffer il faut "ramener" le curseur:

```
buff.rewind() ;
```



4. On exploite maintenant le Buffer:

```
short num = buff.getShort() ;
```



Un exemple de mise en oeuvre de Buffers

Exemple de codage de protocoles utilisant des données de taille fixe: on code ici une chaine de taille maximum 30 ; le protocole demande que l'on indique d'abord la taille de la chaîne

```
public class ProtocolA {
   public final int  TAILLE_MAX  = 30;
   public static final  String  CODAGE = "US_ASCII";
   ByteBuffer buff =  ByteBuffer.allocate(2 + TAILLE_MAX) ;

   public synchronized String  decode(ReadableByteChannel chan)
                              throws IOException{
      buff.clear() ;// Attention ne fait pas le ménage!
      if( 0> chan.read(buff)) return null ;
      buff.rewind() ;
      short num = buff.getShort() ;
      byte[] octets = new byte[num] ;
      buff.get(octets) ;
      return new String(octets, CODAGE) ;
   }
   public synchronized void code (
         WritableByteChannel chan, String chaine)
                              throws IOException {
      buff.clear(); // ne fait pas le menage
      byte[] octets = chaine.getBytes(CODAGE) ;
      short num = (short) octets.length ;
      if( num > TAILLE_MAX) {
         throw new IllegalArgumentException(chaine+ ":too long");
      }
      buff.putShort(num) ;
      buff.put(octets) ;
      buff.rewind() ;
      chan.write(buff) ;
   }
```

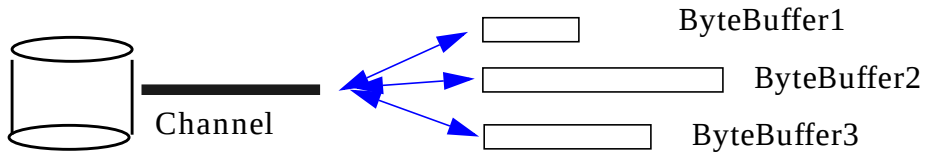
L'exploitation de protocoles spécifiques à d'autres plate-formes non-java est accessible du fait de l'existence de convertisseurs de caractères (`java.nio.charset`-l'utilisation des services de `String` employés ici sont très couteux-) et de `Buffer`s spécialisés avec une autre convention poids-fort / poids-faible (par ex. appel de:
`buff.order(ByteOrder.LITTLE_ENDIAN) ;`)
(on notera dans l'exemple que le `Buffer buff` n'étant pas "nettoyé" on trouvera dans le `Channel` des octets sans signification -la méthode `clear()` n'efface pas les octets dans le Buffer-).

Une mise en oeuvre de la classe précédente :

```
public static void main(String[] args) throws Exception {
   RandomAccessFile rfil = new RandomAccessFile(args[0],"rw") ;
   FileChannel fc = rfil.getChannel() ;
   ProtocolA proto = new ProtocolA() ;
   String res;
   while(null != (res = proto.decode(fc))) {
        System.out.println(res) ;
   }
   rfil.close() ;
   ...
}
```

Ventilation des lectures/écritures dans des Buffers

Les *channel*s `GatheringByteChannel` et `ScatteringByteChannel` acceptent des opérations sur un ensemble ordonné de `ByteBuffers` (on découpe ainsi le traitement des différentes parties d'un protocole).



une seule opération read / write
ventilée sur plusieurs Buffers

Exemple:

```
Charset ascii = Charset.forName("US-ASCII") ; // exception possible
```

utilisation pour un protocole :

```
public void code (GatheringByteChannel gath,
                  short val, String chaine)
                                throws IOException {
      ByteBuffer myShort = ByteBuffer.allocate(2) ;
      myShort.order(ByteOrder.LITTLE_ENDIAN) ;
      CharBuffer myJavaString = CharBuffer.allocate(LG_CHAINE);
      myJavaString.put(chaine) ;
      myJavaString.rewind() ;
      myShort.putShort(val) ;
      myShort.rewind() ;
      ByteBuffer buff = ascii.encode(myJavaString) ;
      gath.write(new ByteBuffer[] {myShort, buff}) ;
   }
```

Les *channels* associé à des fichiers (`FileChannel`), à des Sockets (`SocketChannel`), à des datagrammes (`DatagramChannel`) et à des "pipes" java (`Pipe.SinkChannel` et `Pipe.SourceChannel`) ont des capacités de ventilation.

E/S asynchrones , sélecteurs

Les SelectableChannels permettent des opérations d'entrée/sortie non bloquantes. On enregistre chaque Channel en attente d'une opération auprès d'un Selector. On peut ainsi réaliser des serveurs utilisant un nombre limité de *threads*.

Dans cet exemple c'est un code "client" qui va tenter de contacter en parallèle un certain nombre de serveurs pour obtenir le service echo (service standard sur le port 7).

```
public class Echo {
   public static final int ECHO_PORT = 7 ;
   public static final long TIME_OUT = 60000L ;

   public static void main (String[] args)  throws Exception {
      Selector veille = Selector.open() ;
      ByteBuffer message =
                  ByteBuffer.wrap("hello".getBytes("US-ASCII")) ;

      for(int ix = 0 ; ix < args.length ; ix++) {
         try {
            SocketChannel sckc = SocketChannel.open() ;
            sckc.configureBlocking(false) ;
            SelectionKey skey = sckc.register(veille,
                  SelectionKey.OP_READ| SelectionKey.OP_CONNECT);
            skey.attach(args[ix]) ;// annotation libre!
            sckc.connect(
                  new InetSocketAddress(args[ix], ECHO_PORT)) ;
         } catch (Exception exc) {
            Logger.global.log(Level.SEVERE,args[ix],exc) ;
         }
      }
```

Ici on crée un SocketChannel par nom de machine hôte passé en paramètre. Chaque SocketChannel est rendu asynchrone et est enregistré auprès d'un "sélecteur" . Ce Selector permet une opération bloquante select qui permet d'obtenir un ensemble de *channel*s sur lesquels une opération d'E/S est en attente (ici on s'inscrit pour surveiller les opérations de connection OP_CONNECT et les opérations de lecture OP_READ).

```
        while( veille.keys().size() >0 ) {
           try {
              // appel bloquant
              int nbSelected = veille.select(TIME_OUT) ;
              if(nbSelected == 0) {
                 System.out.println("pas de selection") ;
              }
              Iterator it = veille.selectedKeys().iterator() ;
              while(it.hasNext()) {
                 SelectionKey skey = (SelectionKey) it.next() ;
                 SocketChannel sckc =(SocketChannel) skey.channel();
                 if( skey.isReadable()) {
                    System.out.println(
                              "response from "+skey.attachment()) ;
                    skey.cancel() ;// annulle clef
                    sckc.close() ; // ferme le channel
                 } else if (skey.isConnectable() ) {
                    sckc.finishConnect() ;// conclusions connx
                    System.out.println(
                              "connected to " +skey.attachment()) ;
                    message.rewind() ;
                    sckc.write(message) ;
                 }
               }
           } catch (IOException exc) {
              Logger.global.log(Level.SEVERE,"",exc) ;
           }
}
```

Ici on “sort” de chaque `select` avec un ensemble (`Set`) de clefs qui permettent de retrouver le *channel* correspondant. On peut tester le type d’opération en cours.

- pour une opération de lecture, on invalide la clef et on ferme le *channel*.
- pour une opération de connexion, on termine les opérations de connexion (`finishConnect`) et on envoie le message pour le service `echo`.

Le `read` de lecture n’est pas utilisé ici. Du fait de l’asynchronisme il n’est pas garanti que toute lecture soit complète: il faut donc soigneusement mettre au point tout protocole d’échange.

Compléments

Les compléments techniques suivants constituent une annexe de référence

Compléments sur la linéarisation

Comme un `ObjectStream` mémorise les instances il est possible de forcer la lecture / écriture d'instances sans conservation de référence au moyen des méthodes `writeUnshared/readUnshared` (attention les champs objets d'une instance "unshared" ne sont pas automatiquement considérés comme "unshared").

Dans le cadre d'une personnalisation du mécanisme de linéarisation noter qu'on peut aussi mettre en place un mécanisme de remplacement qui permette de rendre un objet différent de celui qui a été placé dans le flot (voir personnalisation avec méthodes `writeReplace` / `readResolve` sur l'objet linérarisé et autorisation de sécurité correspondante, voir aussi la méthode `resolveObject` spécifique au `Stream`)

On peut aussi lire les champs comme dans `defaultReadObject` (éventuellement sans connaitre a priori la classe cible) on récupère l'ensemble des valeurs de champs dans une instance de `ObjectInputField.GetField` rendue par la méthode d'`InputStream` `readFields`.(voir technique symétrique putFields pour les `OutputStream`)

La personnalisation de bas niveau pour les objets qui implantent l'interface `Externalizable` passe par la définion des méthodes de cette interface (`writeExternal, readExternal`).

Expressions régulières

Le package `java.util.regex` permet de programmer des analyseurs d'expressions régulières:

```
public class IOProps {
   private static Pattern clefValeur ;// thread safe
   static {
      try {
         clefValeur = Pattern.compile(
                  "^\\s*(\\w*)\\s*:(.*)$", Pattern.MULTILINE) ;
      }  catch (PatternSyntaxException exc) {
            assert false : exc ; //si code ci-dessus modifié!
      }
   }// static

   public static Properties propsDepuis(CharSequence seq) {
      Properties props = new Properties() ;
      try {
         Matcher match = clefValeur.matcher(seq) ;
         while(match.find()) {
            props.put(match.group(1), match.group(2).trim()) ;
         }
      } catch(Exception exc) {
         Logger.global.log(Level.SEVERE, "", exc) ;
      }
      return props ;
   }
}
```

Ici on permet d'analyser une `CharSequence` (`StringBuffer`, `String` ou `CharBuffer`). Chaque ligne est de la forme `clef : valeur` . L'expression regulière décrit cette syntaxe (on notera les doubles '\' du fait des contraintes de la syntaxe des chaînes Java).

Ici utilisation d'une assertion . Le code doit se compiler avec l'option:

```
javac -source 1.4 ....
```

et pour valider le contrôle de l'assertion:

```
java -ea ...
```

Utilisation avec une `CharSequence` obtenue à partir d'un Buffer de `java.nio`:

```
public static void main(String[] args) throws Exception {
      FileChannel fc = new
                  RandomAccessFile(args[0],"rw").getChannel() ;
      MappedByteBuffer mbb=
                  fc.map(FileChannel.MapMode.READ_WRITE,0,1000) ;
      CharBuffer cseq =  mbb.asCharBuffer() ;
      Properties props = IOProps.propsDepuis(cseq);
      System.out.println(props) ;
   }
```

(bien entendu la méthode `load` de `java.util.Properties` rend ce service d'une manière plus complète -ceci n'est qu'un programme de démonstration-).

Les références distantes (R.M.I.) 12

Points essentiels

Ce chapitre constitue une introduction technique à R.M.I..

R.M.I. permet de demander des services à des objets distants c'est à dire des objets situés dans une autre J.V.M (éventuellement active sur une autre machine). Bien que demandant un peu d'intendance de mise en place, R.M.I permet une architecture élégante et performante pour réaliser des échanges client-serveur.

La notion de service distant

Imaginons une application bancaire très simple qui gère des comptes. Voici, par exemple, les grandes lignes de la classe `Compte` :

```
public class Compte {

        ....

   public synchronized void depot(BigDecimal val) {
        ....;
   }

   public  synchronized void retrait(BigDecimal val)
                  throws ExceptionDecouvert{
        ...//
   }

   public BigDecimal getSolde() {
        return ....;
   }
}
```

et voici la réalisation du “guichet” qui nous permet d’obtenir une reférence sur une instance de `Compte` lorsqu’on connait le nom du client:

```
public class Banque {
    .... //
   public Compte compteClient (String client) {
      return .....;
   }
}
```

On voudrait maintenant accéder aux services de cette application depuis un poste distant. Ici une instance de Banque “vit” sur le serveur, et les instances de Compte restent également sur cette même JVM. On voudrait donc accéder aux services rendus par ces instances depuis d’autres machines sur le réseau.

Comment le code résident sur l’application cliente va-t-il voir les objets distants?

La définition d'un service distant

Le principe du découplage entre une demande de service et sa réalisation prend ici tout sons sens. Le code demandeur du service du coté client n'a pas à connaître le type effectif de l'objet coté serveur: il va y avoir entre les partenaires un accord de définition d'**interface** de service.

Un autre principe important entre en jeu: la présence du réseau n'est pas transparente. On ne définit pas un service distant de la même manière qu'un service local (rendu dans la même J.V.M): en effet tout appel de méthode implique un transfert de données sur une ligne de communication et cette opération peut échouer pour des raisons liées aux entrée/sortie distantes.

La définition des services distant de Compte et de Banque passe par une définition d'interface particulière:

```
import java.rmi.* ;

public interface CompteDistant extends  Remote{

   public  void depot(BigDecimal val) throws RemoteException ;

   public   void retrait(BigDecimal val)
         throws ExceptionDecouvert, RemoteException ;

   public BigDecimal getSolde() throws RemoteException ;
}
```

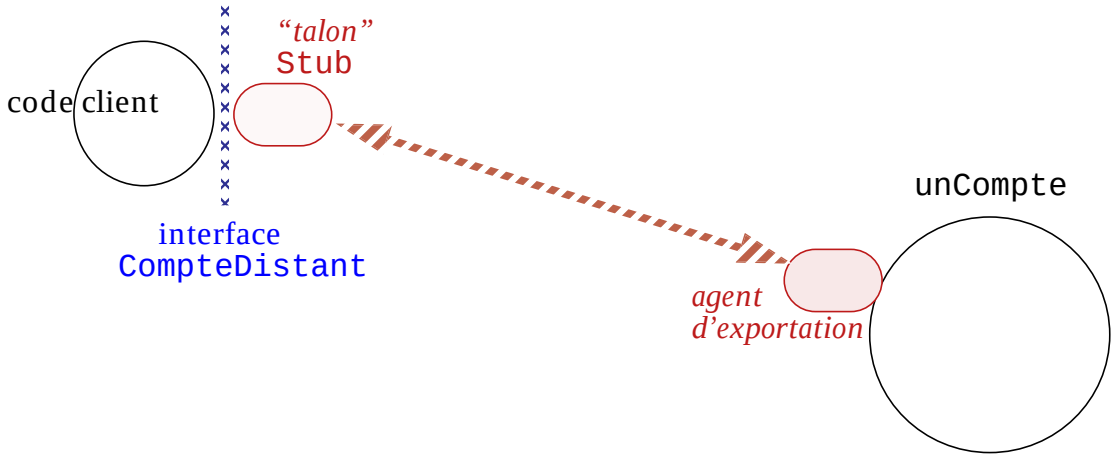
```
public interface BanqueDistante extends Remote{
   public CompteDistant compteClient (String client)
                              throws RemoteException ;
}
```

- chaque interface de service R.M.I (Remote Method Invocation) doit hériter de l'interface de marquage `java.rmi.Remote`. [1]

- chaque méthode du contrat doit impérativement déclarer la propagation d'un `RemoteException` (en plus des exceptions qui leur sont propres).

1. une interface de "marquage" est une interface sans définition de méthode

Principe de la communication entre objets distants



- Du coté serveur un objet "vu de l'extérieur" doit être exporté. Le mécanisme correspondant va mettre en place un *thread* d'écoute pour traiter les demandes de services.
  Sur les anciennes versions de RMI il existait explicitement des classes d'objets chargés de gérer RMI coté serveur: les `Skeleton`s.

- Du coté client un objet particulier appelé "talon" (*Stub*) gère le dialogue avec l'objet distant. C'est ce `Stub` qui est utilisé par l'application cliente au travers de l'interface de service (`CompteDistant` dans l'exemple).

- Les deux codes chargés de gérer les aspect réseaux vont collaborer pour établir des Sockets les maintenir, les rétablir, gérer le protocole d'invocation de méthode distante, et gérer des échanges pour un *garbage-collector* distribué
  Point important: tous les paramètres des méthodes, leur résultat, les exceptions susceptibles d'être propagées sont tous linéarisés. Ils doivent donc tous être d'un type primitif ou `Serializable`

Exportation d'un objet

Le package `java.rmi.server` traite de l'exportation des objets RMI. Une des solutions possibles consiste à définir l'objet exporté comme une sous-classe de `UnicastRemoteObject`:

```
import java.rmi.* ;
import java.rmi.server.* ;

public class CompteExporte extends UnicastRemoteObject
      implements CompteDistant {
   private Compte compte ;

   public CompteExporte (Compte cpt) throws RemoteException {
      compte=cpt  ;
   }

   public void depot(BigDecimal val)throws RemoteException  {
      compte.depot(val) ;
   }

   public  void retrait(BigDecimal val) throws ExceptionDecouvert,
               RemoteException {
       compte.retrait(val) ;
   }

   public BigDecimal getSolde() throws RemoteException{
      return compte.getSolde() ;
   }
}
```

- L'objet exporté implante l'interface de service distante
- Tout constructeur d'objet exporté doit explicitement propager l'exception `RemoteException` (ne serait-ce parce que tous les constructeurs de `UnicastRemoteObject` le font!)

Sur le même principe le code de l'objet Banque :

```
import java.rmi.* ;
import java.rmi.server.* ;

public class BanqueExportee extends UnicastRemoteObject
      implements BanqueDistante {
   private Banque bank ;

   public BanqueExportee()throws RemoteException {
      bank = new Banque() ;
   }

   public CompteDistant compteClient (String client)
                              throws RemoteException{
         // on peut faire partager le même Compte
         // à plusieurs code clients! (bien entendu le programmeur
         // doit gérer cette concurrence d'accès)
      return new CompteExporte(bank.compteClient(client)) ;
   }
}
```

Génération des classes de Stub

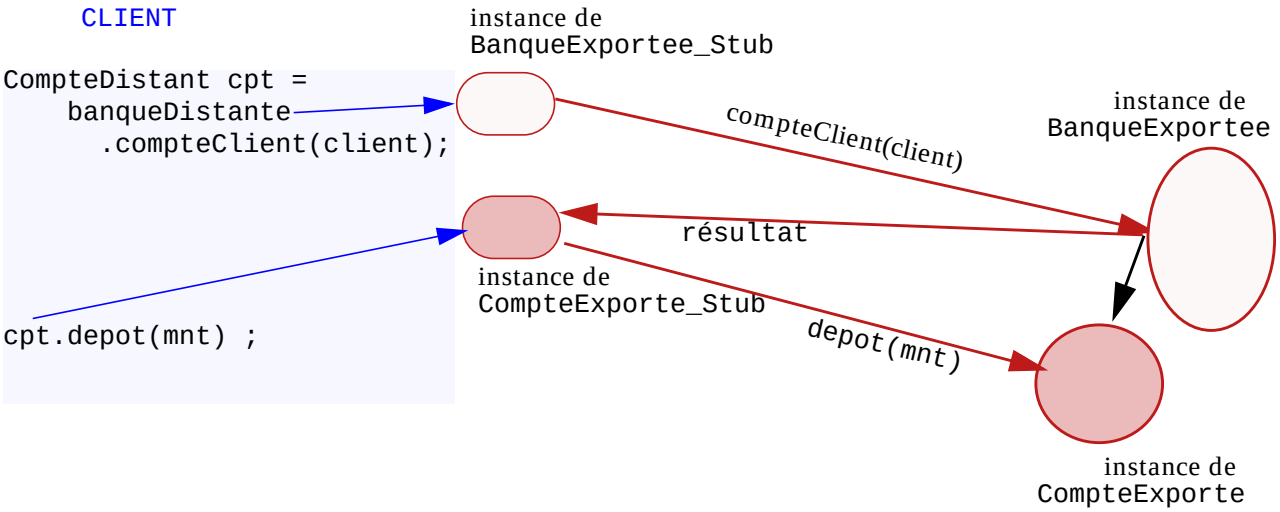
Pour dialoguer avec ces objets exportés il faut disposer d'instance de `Stub`. Les classes correspondantes sont automatiquement générées par l'utilitaire **`rmic`** à partir des fichiers `.class` des objets serveurs.
Exemple (script UNIX):

```
rmic -v1.2 -classpath $CLASSES -d $CLASSES pack.CompteExporte
```

On trouvera ainsi dans le répertoire des classes du package (dans l'arborescence située sous le répertoire désigné par `$CLASSES`) un fichier automatiquement généré et de nom `CompteExporte_Stub.class`.

(de la même manière on générera `BanqueExportee_Stub.class`).

Les instances de Stub



Prenons par exemple un code client qui réalise les opérations suivantes:

```
BanqueDistante banqueDistante ;
..... // codes divers
CompteDistant cpt = banqueDistante.compteClient(client);
```

La référence `banqueDistante` est typée par l'interface de service, mais l'objet qui va rendre le service est de type effectif `BanqueExportee_Stub`.
Ce talon va appeler l'objet serveur, lui signifier l'appel de la méthode et lui passer l'argument linéarisé.
L'objet serveur va renvoyer en résultat une instance de `CompteExporte_Stub` qui représente la référence distante.

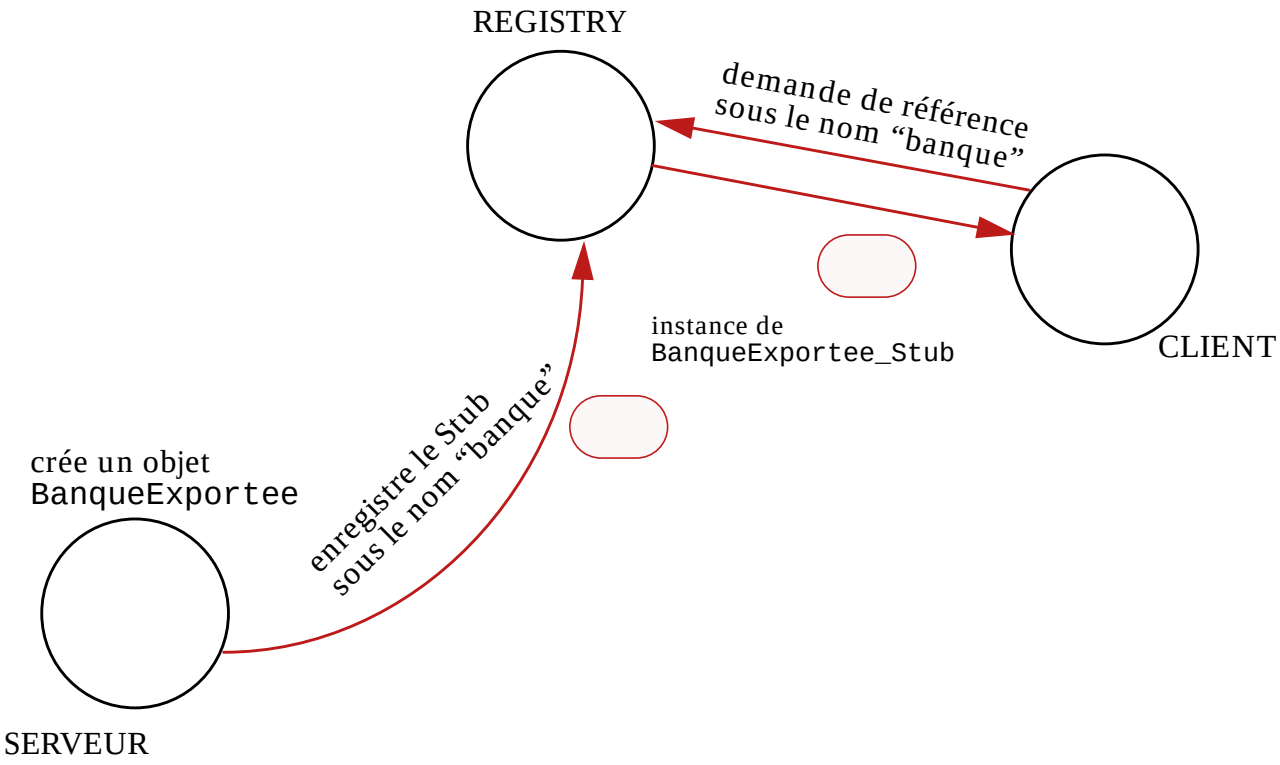
Cette référence distante est vue par le code client comme de type `CompteDistant` et c'est sur cette référence que seront invoquées les méthodes correspondantes:

```
cpt.depot(mnt);//le talon communique avec l'objet correspondant
```

Mais comment a-t-on amorcé ce processus, comment a-t-on obtenu la première reference distante sur un objet `BanqueExportee` ?

Annuaire d'objets distants

Pour obtenir une référence sur un objet distant l'application cliente doit s'adresser à un agent "bien connu", pour lui demander un objet référencé par un nom. Inversement une application serveur qui souhaite rendre un service distant doit appeler cet annuaire pour enregistrer une référence sous ce nom (dans l'exemple nous avons choisi le mot-clef "banque").
En RMI cette application d'intermédiation est appelée *registry*.



L'application de référence fournie avec le SDK est **rmiregistry** :

- Elle doit être lancée sur le même hôte réseau que le serveur.
- Elle "écoute" par défaut sur le port 1099
- On peut la lancer avec la commande:

```
rmiregistry [num_port]
```

Le serveur: enregistrement auprès du Registry

Voici un code de serveur très simple:

```
...
import java.rmi.* ;
public class Serveur {
   public static void main (String[] args) throws Exception {
      BanqueExportee bank = new BanqueExportee() ;
      Naming.rebind("banque", bank) ;
   }
}
```

- On créé un objet `Banque` serveur
- On l'enregistre auprès du *registry* sous le nom "banque"
  Cet appel a pour effet d'envoyer une instance de `BanqueExportee_Stub` au *registry*.
  Il y a deux méthodes statiques possibles dans la classe `Naming` : `bind(..)` pour un enregistrement définitif, `rebind(..)` pour des re-enregistrements successifs.

On notera que le programme ne s'arrête pas: il y a en effet maintenant un *thread* d'écoute à l'attente des demandes de services sur l'objet exporté. Utiliser `UnicastRemoteObject.unexportObject()` pour retirer un objet du *runtime* RMI.

Pour une forme plus dévelopée de l'argument du `bind` voir page suivante.

Le client: demande au Registry

Un code de client très simple :

```
...
import java.rmi.* ;
public class Client {
   public static void main (String[] arg) throws Exception {
         ........
      BanqueDistante bk =
         (BanqueDistante) Naming.lookup("rmi://"+hote+"/banque") ;
      CompteDistant cpt = bk.compteClient(nomClient) ;
      cpt.depot(depot) ;
      cpt.retrait(retrait) ;
      ........;
   }
}
```

- La demande de référence distante initiale au *registry* se fait au moyen de la méthode `Naming.lookup()` .
  Comme cette méthode rend un résultat de type `Object` noter la nécessiter d'opérer une projection de type (*cast*).

- L'argument de désignation est une chaîne en format URL de la forme:

`rmi://hote:port/clef`

  - `hote` est la désignation de la machine hôte sur laquelle tourne le *registry*
  - la désignation du port est optionnelle (si le *registry* écoute sur un port différent du port par défaut 1099)
  - `clef` est le mot-clef sous lequel est enregistré le service.
  - on notera que cette désignation est aussi valable pour le `bind`

```
//code coté serveur
Naming.rebind("rmi://localhost:6666/banque", bank );
// ici registry écoute sur port 6666
```

Check-list pour une mise en oeuvre simple

Le cadre de notre exemple suppose que les classes suivantes soient accessibles dans le `classpath` des applications:

`SERVEUR`:

```
Banque.class                    CompteDistant.class
BanqueDistante.class            CompteExporte.class
BanqueExportee.class            CompteExporte_Stub.class
BanqueExportee_Stub.class       ExceptionDecouvert.class
Compte.class                    Serveur.class
```

Remarquer la nécessité de disposer des classes de `Stub` : le `bind` envoie au *registry* une instance de ce type.

`RMIREGISTRY`:

```
BanqueDistante.class            CompteDistant.class
BanqueExportee_Stub.class       CompteExporte_Stub.class
```

Le cas est ici particulier pour deux raisons:

- `rmiregistry` décode les objets qui lui sont passés et donc a besoin des classes et interfaces correspondantes au type des `Stub`s. Ce comportement est particulier à cette implantation de référence du service de *registry*.

- Dans le cas de chargement dynamique des classes de Stub (voir annexe) il faut, au contraire, éviter d'avoir de telles classes accessibles dans le `classpath` de `rmiregistry`.

`CLIENT`:

```
BanqueDistante.class            CompteDistant.class
BanqueExportee_Stub.class       CompteExporte_Stub.class
Client.class                    ExceptionDecouvert.class
```

Ici c'est la présence des classes de `Stub` qui peut surprendre. Dans de nombreux cas de déploiement, comme pour les Applets, ceci ne pose aucun problème puisque c'est le serveur qui déploie le code client. Dans d'autres cas on pourra s'orienter vers un chargement dynamique de ces classes (voir annexe).

Exercice: mise en place de R.M.I.

Exercice :

- On définira un service distant de la manière suivante :

```
package rmi ;
import java.rmi.* ;
import java.sql.* ;

public interface TimeService extends Remote {
        Timestamp[] echo(Timestamp arg) throws RemoteException;
}
```

Ici le service demandé est analogue à celui vu dans “Mini-exercice:”, page 188. On envoie un `Timestamp` et on en récupère deux. Le “nom” du service (tel qu’il sera indexé par le *Registry*) sera “timestamp”.

Faire l’exercice en plusieurs phases :

- d’abord en lançant le test en local et en partageant toutes les classes entre client, serveur et *registry*
- ensuite en séparant soigneusement les classes visibles par le client, le serveur, le *registry* (toujours en local).
- Pour les très rapides (ou ceux qui révisent) : faire fonctionner le service entre plusieurs machines avec téléchargement dynamique des classes.

Compléments

Les compléments techniques suivants constituent une annexe de référence

Telechargement dynamique des codes de classe

Un partenaire d'un échange RMI peut se trouver dans une situation où il ne dispose pas d'un code de classe:

- parcequ'une classe de `Stub` n'a pas été installée localement
- parceque le type effectif d'un objet accompagnant un appel de méthode n'est pas connu localement. Du fait du polymorphisme on peut avoir en effet un type *runtime* différent du type déclaré dans le "contrat" de l'interface `Remote`.

On notera que cette incertitude peut exister pour chacun des partenaires (client ou serveur -de fait la distinction traditionelle entre client et serveur peut devenir peu pertinente: un "serveur" pouvant prendre l'initiative d'un échange vers un objet exporté par le client-).

Contexte de sécurité

Le premier point est que toute J.V.M qui télécharge dynamiquement du code doit mettre en place un `SecurityManager` (s'il n'est pas déjà présent -cas des Applets par ex.-):

```
System.setSecurityManager(new SecurityManager());
//ou par option lancement de la J.V.M.
```

La présence d'un environnement de sécurité nécessite une mise en place d'une politique de sécurité:

```
java -Djava.security.policy=myrmi.policy pack.Client .....
```

Ce fichier policy contenant une entrée de type :

```
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`).

Mise en place du téléchargement des classes

Les objets impliqués dans des échanges R.M.I. contiennent une annotation qui permet d'indiquer une URL d'origine de la classe correspondante. Le `ClassLoader` local qui "désérialize" l'objet peut prendre en compte cette URL pour charger la classe en fonction du protocole indiqué dans l'URL.

De manière pratique le protocole le plus utilisé est **http**. Ce qui suppose la mise en place d'un serveur http pour exporter le code des classes.

Pour annoter correctement les objets et `Stub`s impliqués :

- Soit fixer la propriété `java.rmi.server.codebase` dans le contexte de la JVM exportatrice.
- Soit fixer un ensemble d'URL accessibles et les passer à un `URLClassLoader`. Exemple :

```
public static void main(String args[]) throws Exception {......
   URL[] tbURL =  new URL[args.length]  ;
   for (int ix = 0 ; ix < args.length; ix++) {
      tbURL[ix] = new URL(args[ix]) ;
   }
   URLClassLoader classLoad = new URLClassLoader( tbURL) ;
    Remote ss =  (Remote) classLoad.loadClass(nomClasseRacine).
                              newInstance();
                    // ici la classe "racine" a un constructeur sans paramètre
//attention les classes dépendantes ne doivent pas être vue du classpath local
    Naming.rebind("Server", ss) ;
  }
}
```

Dans les deux cas il est souhaitable d'organiser un déploiement dans des jars et d'utiliser des URLs comme:
`jar:http://server:8080/mesclasses.jar!/`

Attention: dans ce cas si on utilise `rmiregistry` comme *registry* prendre soin de lui retirer toute visibilité directe sur les classes concernées (sinon il reconstituera des annotations erronées au moment où il envoie le `Stub` au client).

Personnalisation des mécanismes sous-jacents à RMI

La classe `java.rmi.server.UnicastRemoteObject` dispose de constructeurs ou de méthode de classe qui permettent :

- de fixer le numero de port utilisé pour “exporter” l’objet (utile dans un contexte sous `SecurityManager`).
- de personnaliser les `Sockets` sous-jacentes (via `RMIServerSocketFactory` et `RMIClientSocketFactory`). On peut ainsi par exemple mettre en place des sockets UDP, des `Sockets` SSL -voir package `javax.net.ssl`-), ou des `Sockets` avec compression des données, etc. voir:
  `docs/guide/rmi/socketfactory/index.html`

Pour une liste des `Properties` de configuration voir :
`docs/guide/rmi/spec/rmi-properties.html`.

Par ailleurs RMI est automatiquement capable de transformer une requête bloquée par un pare-feu en appel HTTP `POST`.

Serveurs dormants, serveurs résilients

Les objets exportés ne sont pas nécessairement des instances situées dans un serveur actif. On peut tout à fait déclarer un objet pour l''exportation" et ensuite arréter la JVM serveur.
Lorsqu'un client demande une référence l'objet est resuscité au sein d'une JVM active.

Ces objet particuliers sont `Activatable` et un démon système particulier **`rmid`** sert à enregistrer ces objets et à les activer (notons que rmid joue aussi le rôle de *registry*).

Voir `docs/guide/rmi/activation.html`

JINI

(voir `http://www.jini.org`)

JINI permet l'organisation de "services spontanés" sur un réseau.

C'est un ensemble de protocoles et de règles de programmation qui permettent de rechercher un service (ou de le "publier"), de télécharger un mandataire (*proxy)* de service (code java qui gère un dialogue avec le service, ou partie de service qui s'exécute sur le client).

L'utilisation de ces services par des codes clients est soumis à des règles de comportement (*bail* de ressource sur le serveur, événements distribués, transactions distribuées).

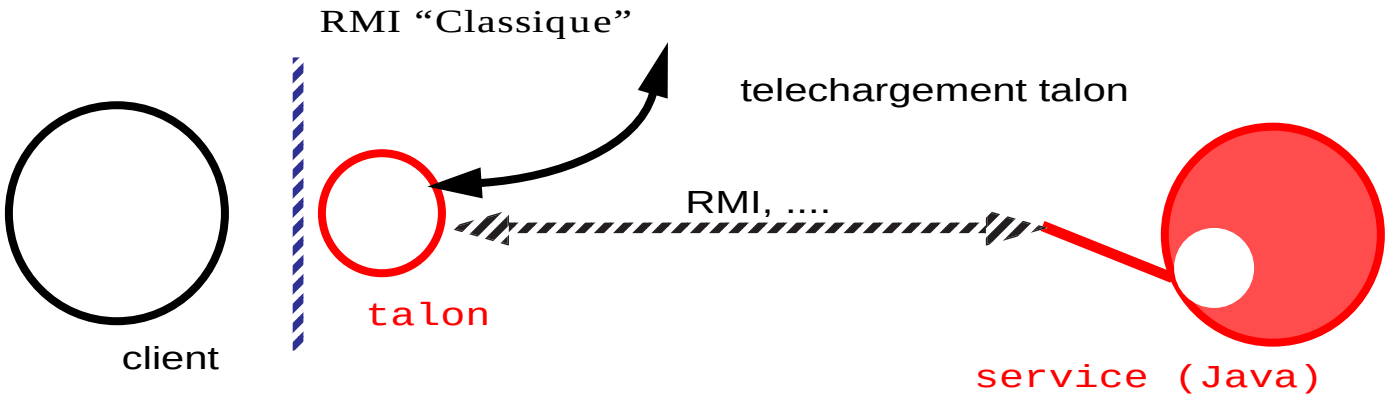
Protocoles de recherche et découverte

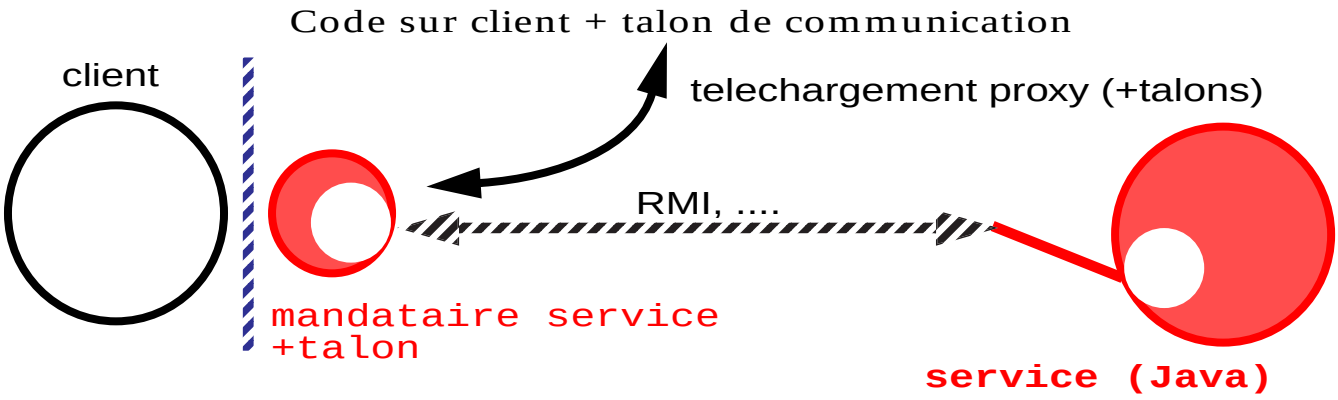
- `Unicast Discovery Protocol` : on se connecte à un *`ServiceRegistrar`* connu (URL Jini, messages TCP): pour enregistrement d'un service, recherche d'un service, enregistrement d'une consigne (évt. distribué).

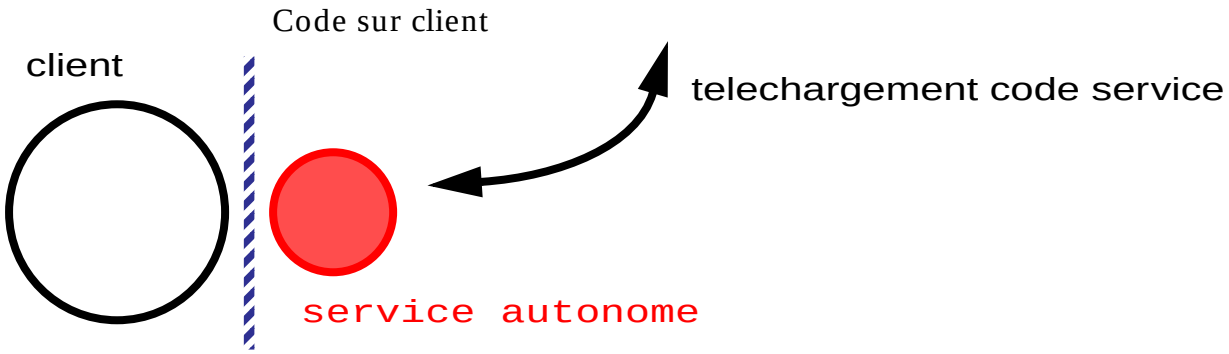
- `Multicast Request`: demande de recherche de *ServiceRegistrar* diffusée en multicast.

- `Multicast Announcement`: un *ServiceRegistrar* annonce son existence sur le réseau.

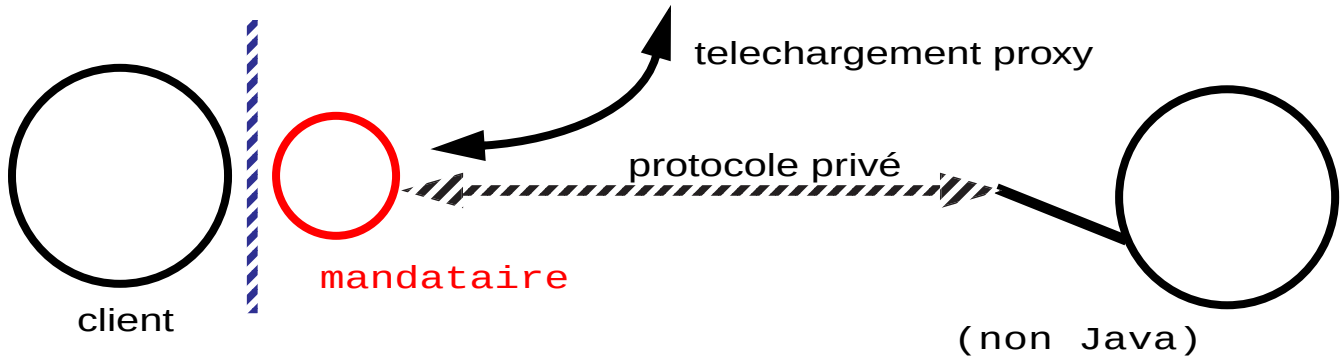
Utilisation des Services



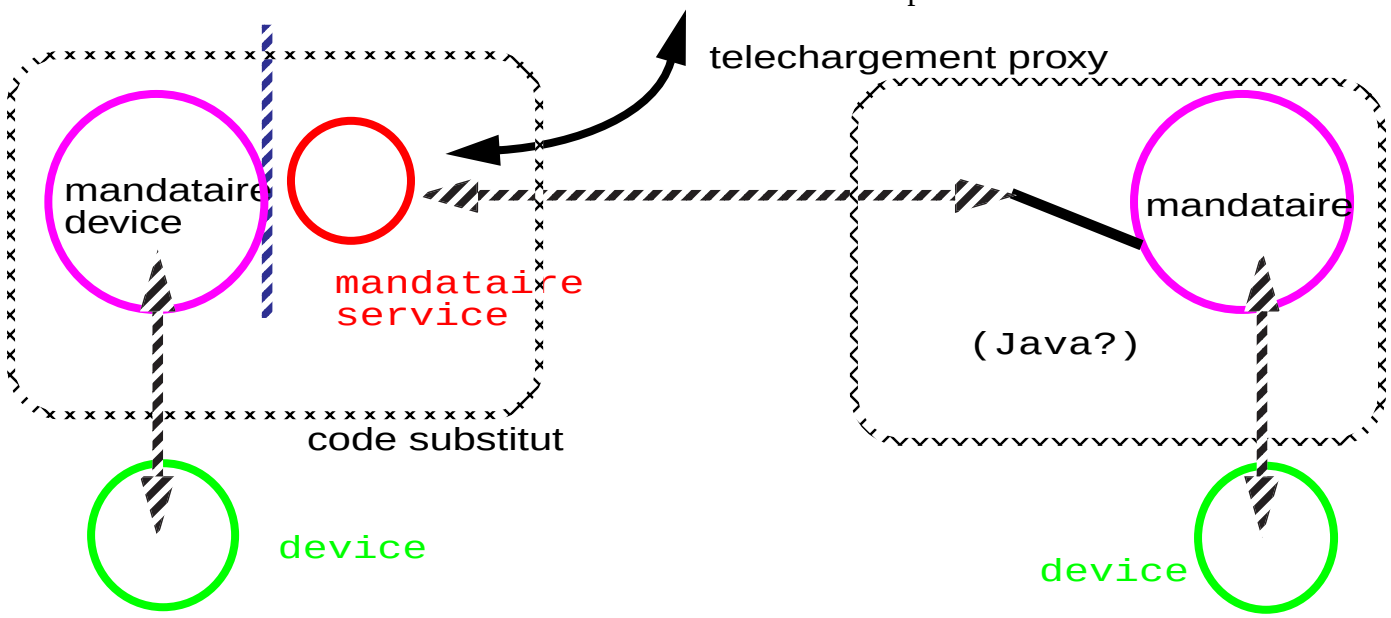




Talon d'accès à un service non-Java



Codes Java servant de "frontaux" à des dispositifs matériels



L'accès aux bases relationnelles (J.D.B.C) 13

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

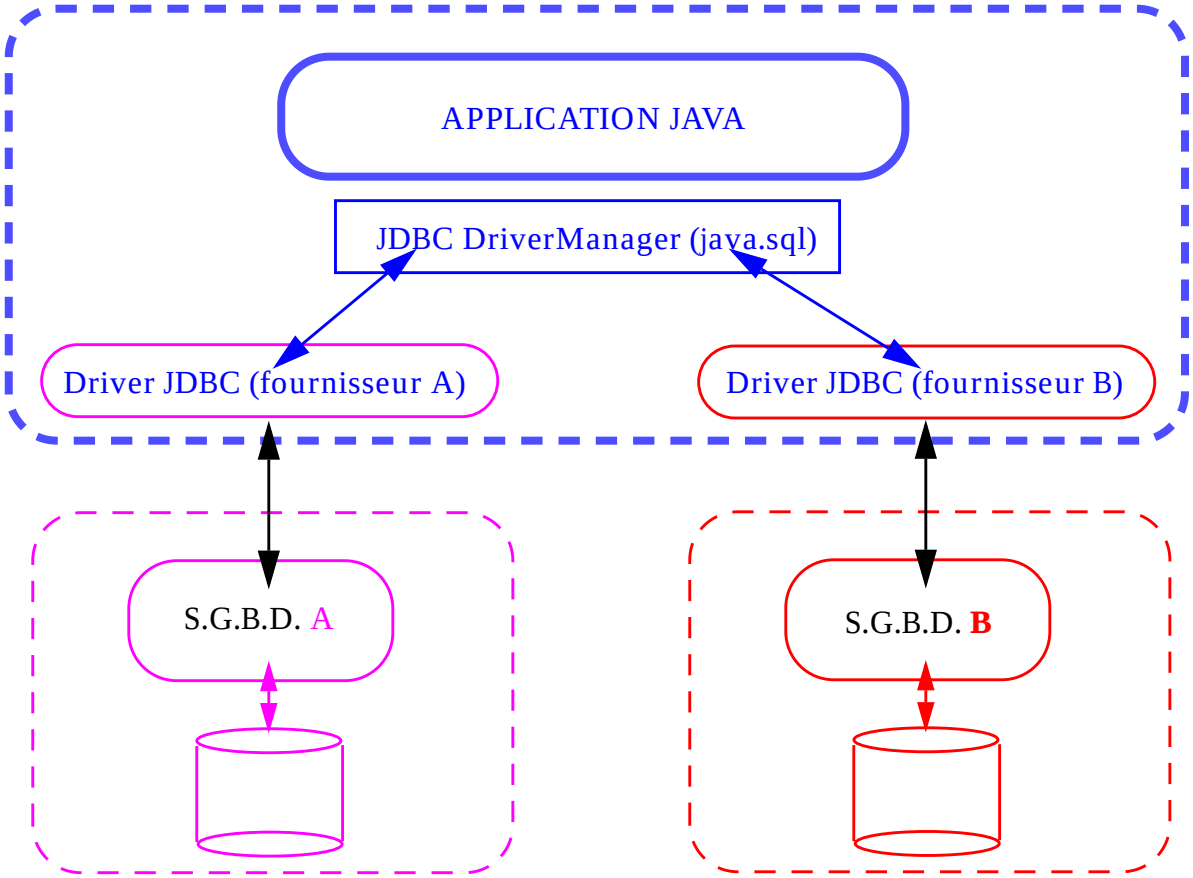
Ce chapitre constitue une introduction technique à JDBC:

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

Le package `java.sql`

Le package `java.sql` permet de réaliser des applications qui exploitent des bases de données relationnelles. L'accès aux bases se fait en SQL au travers de modalités standardisées (JDBC). En ce sens JDBC est typique des API d'accès: java.sql définit essentiellement des interfaces d'accès qui sont réalisées par des pilote (*drivers*).

Une application Java qui fait appel à JDBC s'appuiera sur un petit nombre de classes standard (comme le `DriverManager`) qui permettront l'utilisation de classes complémentaires livrées avec les pilotes spécifiques à une base..



Liste indicative des pilotes : `http://java.sun.com/products/jdbc/jdbc.drivers.html`

Types de drivers JDBC

Le choix des pilotes dépend des caractéristiques de la base et du type d'architecture système retenue :

- architecture à deux parties : l'application communique directement avec le système de gestion de base de données.

- architecture à trois parties : l'application "cliente" fait traiter ses requêtes par une application "serveur" qui gère les modalités et la cohérence des appels à la base de données.
  Le protocole client/serveur peut très bien ne contenir aucune référence à SQL.

On classe habituellement les pilotes en 4 catégories:

1. Pont JDBC-ODBC : permet de transcrire les appels JDBC en appels à un driver ODBC (les deux modèles sont assez proches). Les driver ODBC sont réalisés en code natif.

2. Accès direct à l'API de la base: le pilote est réalisé en code natif sur l'API d'accès à la base de données.

3. Driver deux parties JDBC-Net: le pilote est en pur Java et communique avec un correspondant capable de dialoguer directement avec la (ou les) base(s).

4. Driver pur Java sur protocole spécifique: le pilote retranscrit les appels JDBC en appels du protocole distant spécifique à la base.

Modalités d'une session JDBC

La réalisation d'un accès JDBC aux bases de données suit toujours le même schema :

- Désignation de la ressource cible. Cette désignation se fait au travers d'une chaîne de description d'une URL et permet d'activer un pilote particulier. Le DriverManager permet cette activation à condition que le programmeur ait veillé à lui faire connaître un pilote adapté.

- Obtention d'un objet de type `Connection`.
  Permet de gérer une session de consultation de la base.

- Obtention d'un objet de type `Statement`.
  Dans le cadre d'une `Connection` on obtient des objets de ce type pour formuler des requêtes SQL à la base.

- Exploitation des résultats : si la requête est de type `SELECT`, les objets `ResultSet` permettent de recupérer les données résultantes.

Enregistrement des pilotes

Il y a deux méthodes principales pour faire connaître un pilote JDBC à l'objet `DriverManager`:

- Chargement explicite de la classe principale du pilote.

  Les spécifications JDBC précisent que lorsque le ClassLoader charge une classe qui réalise l'API décrite par `java.sql.Driver`, celle ci doit créer un objet du type correspondant et l'enregistrer auprès du DriverManager.

  Dans ce cas il faut:

  - Connaître le nom complet de la classe Driver: par ex. `com.imaginary.sql.msql.MsqlDriver`
  - Faire en sorte que cette classe (ou une archive .jar la contenant) soit dans le CLASSPATH de l'application.

```
public class LanceurJDBC {
        public static void main(String[] args) throws Exception{
                Class.forName(args[0]);
                // nom de la classe passée en paramètre d'appel
                // ou de tout autre manière
                ....
        }
}
```

- Communication d'une liste de classes Driver au travers des arguments d'appels de Java:

```
java -Djdbc.drivers=com.acme.ThatDriver:fr.gibi.MonPilote
```

Le DriverManager peut interroger le système pour connaître la liste des pilotes disponibles et les charger.

Désignation de la ressource

Pour établir une connexion à une base on va adresser au `DriverManager` une chaîne d'identification de la ressource recherchée.

Cette chaîne doit suivre une syntaxe standard qui suit la syntaxe des URL:

```
jdbc:sous_protocole:parametres
```

“sous_protocole” désigne un type spécifique de mécanisme pouvant être réalisé par un ou plusieurs pilotes. Le contenu des paramètres dépendent de ce “sous_protocole”. Il convient donc de suivre le mode d'emploi indiqué par les pilotes pour savoir désigner une ressource d'une manière adaptée: c'est la rédaction correcte de cette url qui permettra au pilote de se reconnaître.

En effet le `DriverManager` essaie, dans l'ordre, tous les pilotes enregistrés en leur passant l'url demandée. Le premier pilote qui sait établir la connexion emporte la décision.

Voici, par exemple, une chaîne de caractères représentant une url pour un pilote de nom “msql” qui établit une connexion réseau sur une machine serveur sur un numéro de port donné:

```
jdbc:msql://monserveur:6666/MaBase
```

ceci étant fabriqué par exemple par l'instruction:

```
String url = “jdbc:msql://” +nomserveur + “:” + port +”/” + base ;
```

Obtention de la connexion

Pour obtenir une instance d'accès à la base conforme à `java.sql.Connection` on doit appeler:

```
Connection conx = DriverManager.getConnection(chaineUrl) ;
```

Cet objet permet d'obtenir des informations sur la base et, surtout, permet d'établir des contextes de requêtes.

D'autres formes de `getConnection` existent; si aucun pilote n'est trouvé le résultat sera null, si une anomalie s'est produite au regard des exigences de la base une `SQLException` sera propagée.

Obtention d'un contexte de requêtes

Des objets "contexte de requête" (`Statement`) peuvent être obtenus dans le cadre d'une connexion. Un objet Statement permet de traiter une seule requête SQL statique en même temps. Si l'on veut traiter plusieurs requêtes en parallèle il faut ouvrir plusieurs `Statement`.

```
// création d'un contexte de requête
Statement stmt = null ;
try {
        stmt = conx.createStatement() ;
} catch (SQLException exc) {
        // traiter exception
}
```

L'objet `Statement` permet d'exécuter divers type de requêtes, ainsi , par exemple, pour une requête modificative de la base (INSERT, UPDATE, DELETE):

```
//
int nombre = stmt.executeUpdate(
        "DELETE from Clients WHERE id = '16899023455'");
```

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

Exploitation de resultats

Dans un contexte Statement la méthode executeQuery permet de lancer une requête rendant des enregistrements (SELECT):

```
ResultSet res = stmt.executeQuery("SELECT * from Clients") ;
```

L'objet de type ResultSet permet d'obtenir des informations sur le résultat et de parcourir les enregistrements:

```
while (res.next()) {
        String nomClient = res.getString(2) ;
        String id = res.getString(1) ;
        table[ix++] = new Client(id, nomClient) ;
}
```

Les colonnes de l'enregistrement sont numérotées à partir de 1. Pour chaque colonne on peut récupérer le résultat dans un type Java approprié en utilisant une des méthodes getXXX() -voir table page suivante-. Il est également possible de rechercher une colonne par son nom plutot que par son numéro d'ordre.

Les `ResultSets` permettent également des récupérations par blocs de N enregistrements:

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

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()` -si le service est accessible pour le driver considéré-

Correspondance des types de données SQL en Java

Quelques types Java standard pour la correspondance avec divers types SQL courants.

Table 1: 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

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. :

Exemple

```
 public void prepStatement(ListeReservations liste){
          PreparedStatement prepStmnt = msqlConn.prepareStatement(
"UPDATE Flights SET numAvailFCSeats = ?  WHERE flightNumber = ?" );
          for ( .... ) { //parcours de la liste
                    Reservation rsv = .... ;
                    prepStmnt.setInt(1,
                      (Integer.parseInt(rsv.numAvailFCSeats) - 1));
                    prepStmnt.setLong(2, rsv.FlightNo);
                    int rowsUpdated = prepStmnt.executeUpdate();
                    ...
          }
 }
```

Si la requête préparée avait été basée sur un SELECT c'est un `executeQuery` qu'il aurait fallu appeler.

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`. .

Exemple de requête 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){
        Logger.global.log(Level.SEVERE,"return_seats", SQLEx);
}
```

On notera la syntaxe de désignation de la procédure (type ODBC)

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`.

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() ;
```

Exercice: mise en place de J.D.B.C

Exercice :

- L'animateur mettra à votre disposition une base de données, son driver et vous donnera une description des tables disponibles.

  Réaliser une interface graphique qui vous permette de taper une requête SELECT et de consulter les résultats de votre requête.

ANNEXES 14

Programmation dynamique 15

Points essentiels

La classe `Class` et le package `java.lang.reflect` : les bases de la programmation dynamique.

La “découverte” et l’utilisation au *runtime* d’objets d’un type “inconnu”. La définition de protocoles auquels doivent adhérer ces objets inconnus (convention de `Bean`) permet de créer des outils capables de manipuler dynamiquement des catégories d’objets.

Pourquoi une programmation sur des classes découvertes au runtime?

Java étant un langage compilé, les codes que l'on écrit opèrent sur des classes connues au *compile-time*. Il est pourtant possible d'écrire des programmes qui opèrent sur des classes "inconnues" du compilateur:

- Du fait du polymorphisme des opérations définies par des types connus du compilateur (classes ou interfaces) peuvent en réalité opérer sur des objets ayant un type effectif non apparent. Les classes correspondantes peuvent même avoir été découvertes et chargées dynamiquement au *run-time* .

- Au delà de l'utilisation d'une API "connue" il peut s'avérer utile de découvrir l'API d'une classe au *run-time* et, mieux encore, de l'utiliser. C'est en particulier le cas d'outils écrits en Java et chargés de mettre en oeuvre des codes Java écrits par ailleurs: compilateurs (vérification de la compatibilité d'une A.P.I.), génération de protocoles (comme la linéarisation des objets), outils de déploiement, assemblages interactifs de composants, composants graphiques génériques s'adaptant à des "modèles" prédéfinis, etc.

La classe `java.lang.Class`

Pour pouvoir opérer à un niveau interprétatif sur un objet on a besoin d'une instance de la classe `java.lang.Class` représentant la classe de l'objet. Une telle instance peut s'obtenir par :

- chargement dynamique d'une classe au travers d'un ClassLoader. On doit alors fournir une chaîne donnant le nom complet de la classe (hiérarchie de package + nom de classe). Par ex.:

```
Class laClasse = Class.forName(nom) ;
```

- obtention de la classe d'une instance (obtenue par différents moyens, par exemple un transfert distant):

```
Class laClasse = instance.getClass() ;
```

- consultation d'un champ statique connu au compile-time:

```
Class maClasse = MaClasse.class ;
Class tabCharClass = char[].class ;
```

cas particulier des scalaires primitifs :

```
Class typeInt = int.class ;
Class typeInt = Integer.TYPE ;
```

(ces classes représentant des primitifs servent à repérer par ex. les types des arguments d'une méthode: on ne peut pas faire des opérations "objet" avec)

Obtention des informations sur la classe

A partir d'une instance de `Class` (qui ne répond pas à la condition `isPrimitive()`) on peut obtenir des informations sur les constituants. Par exemple :

- quels sont les champs déclarés dans la classe? :

```
Field[] champs = maClasse.getDeclaredFields() ;
```

si la classe représente une tableau (`isArray()` répond `true`), il est possible de connaître le type des composants :

```
Class typeComposants = classeTableau.getComponentType();
```

- quelles sont les méthodes définies dans la classe?

```
Method[] méthodes = maClasse.getDeclaredMethods() ;
```

- quels sont les constructeurs définis dans la classe?

```
Constructor[] constructeurs = maClasse.getDeclaredConstructors();
```

L'utilisation des classes correspondantes (`Field`, `Method`, `Constructor`,...) relève du package `java.lang.reflect`.

Le package `java.lang.reflect`

La manipulation des champs, méthodes, constructeurs peut poser des problèmes de droit d'accès. Pourtant certains mécanismes (voir par exemple la linéarisation) doivent pouvoir agir indépendamment des privilèges de responsabilité (`private`, `protected`, etc.):

- Membres et constructeurs de la classe dérivent tous de `java.lang.reflect.AccessibleObject` qui dispose d'une méthode `setAccessible(boolean)` qui permet de passer outre aux privilèges de responsabilité.

- Bien entendu n'importe quel code ne peut réaliser ces opérations sans disposer d'une autorisation de sécurité adequate: `java.lang.reflect.ReflectPermission "suppressAccessChecks"`

Les champs

A partir d'une instance de **`Field`** on peut obtenir :

- son nom:

```
String nom = champs[ix].getName() ;
```

- son type

```
Class type = champs[ix].getType() ;
```

- d'autres informations comme les modificateurs utilisés.

On peut également consulter la valeur de ce membre (ou la modifier). Les méthodes get/setXXX permettent de lire ou modifier une valeur sur une instance passée en paramètre :

```
Class laClasse = instance.getClass() ;
Field leChamp0 = laClasse.getDeclaredFields() [0] ;
.... // obtention informations de type
.... // fixation evt. des droits d'accès
...../// création d'un objet de ce type: valeurObjet
leChamp0.set(instance, valeurObjet) ;
```

Si le champ est statique le premier argument est ignoré (il peut être `null`).

Les méthodes

A partir d'une instance de **`Method`** on peut obtenir :

- son nom:

```
String nomMeth = méthodes[ix].getName() ;
```

- le type de son résultat

```
Class typeRes = méthodes[ix].getReturnType() ;
```

- les types des paramètres

```
Class[] typeparms = méthodes[ix].getParameterTypes() ;
```

- d'autres informations comme les modificateurs utilisés ou les exceptions déclarées.

On peut également invoquer la méthode dynamiquement sur cet objet en lui passant les arguments appropriés (et éventuellement en récupérant une `InvocationTargetException` qui chaîne une Exception provoquée par l'invocation sous-jacente).

```
Class laClasse = instance.getClass() ;
Method laMeth0 = laClasse.getDeclaredmethods() [0] ;
.... // fixation evt. des droits d'accès
.... // obtention informations des type des paramètres
.....// création d'un tableau d'arguments
     //** pour chaque type primitif on passe une instance
     //** de la classe d'encapsulation correspondante
     //** par ex. un Integer pour int.TYPE
try {
   laMeth0.invoke(instance, tableauArguments) ;
} catch (Exception xc) {}
```

Si la méthode est statique le premier argument est ignoré (il peut être `null`).

Les constructeurs

A partir d'une instance de **`Constructor`** on peut obtenir :

- les types des paramètres

```
Class[] typeparms = constructeurs[ix].getParameterTypes() ;
```

- d'autres informations comme les modificateurs utilisés ou les exceptions déclarées.

La création dynamique d'une instance peut se faire par `maClasse.newInstance()` (s'il ya un constructeur sans paramètres) ou par l'invocation de newInstance :

```
try {
   constructeur.newInstance(tableauArguments) ;
} catch (Exception xc) {}
```

Les tableaux

La classe `java.lang.reflect.`**`Array`** permet des accès particuliers aux objets qui sont des tableaux (elle permet également la création de tableaux à une ou plusieurs dimensions).

Pour accéder en lecture ou en écriture les méthodes set/getXXX sont dotés d'un argument représentant l'index dans le tableau. Les méthode statiques `newInstance` permettent de créer des tableaux d'un type donné:

```
public static Object[] subArray(Object[] el, int idx, int size) {

         ...// verification validité paramètres

         Object[] res =
                   (Object[])java.lang.reflect.Array.newInstance(
                      el.getClass().getComponentType(), size);

         System.arraycopy(el, idx, res, 0, size);

        return res;
}
```

Point intermédiaire : programmation dynamique

Mini-exercice (pour ceux qui révisent):

- Reprendre l'interface graphique Question/Réponse et sur le modèle du code qui analysait les demandes de calcul sin/cos faire un code qui utilise les services statiques d'une classe (comme Math ou StrictMath) pour leur demander l'exécution d'un calcul:

```
atan 3.45 // calcule arc-tangente de 3,45
```

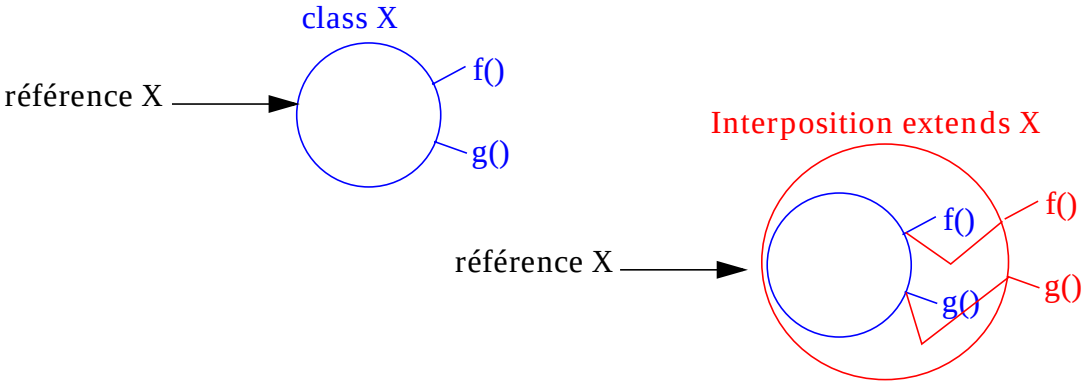
essayer de perfectionner en permettant des arguments en nombre et types variés et en internationalisant :

```
racinecarrée 6,5859
```

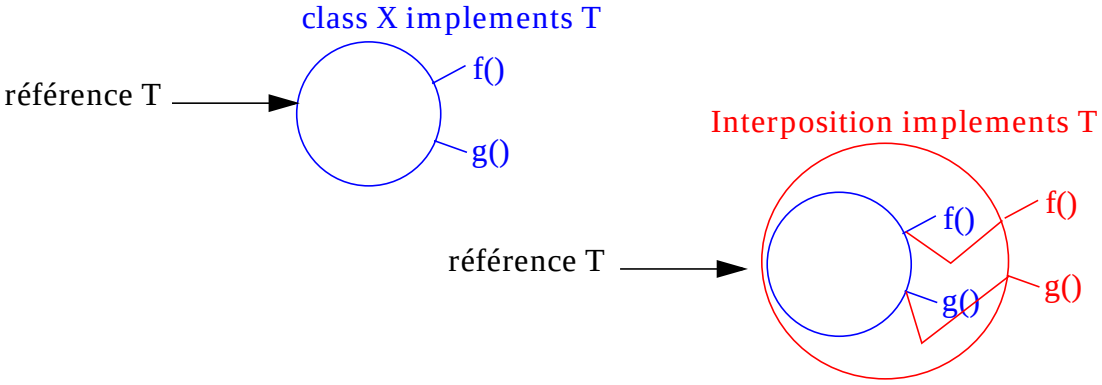
Mandataires dynamiques

Un mandataire (*proxy*) est un objet qui s'interpose devant un autre objet pour intercepter les messages et modifier la façon dont un service est rendu. En Java cette interposition peut se réaliser :

- soit avec un objet d'un type qui sous-classe le type de la référence demandée et qui spécialise ses méthodes (pour evt. déléguer à un objet du type demandé: Voir *Introduction: un cas particulier d'héritage/délégation*, page 167.)



- soit avec un objet qui implante une interface correspondante au type demandé et qui délègue éventuellement à une instance de l'objet initial.



Dans ce dernier cas les mandataires dynamiques (*dynamic proxy*) permettent de générer dynamiquement des objets d'un type dont la classe effective est définie au *runtime*. Il peut y avoir plusieurs raisons pour opérer dynamiquement dont, en particulier, le fait que l'interposition ne s'adresse pas forcément à un objet d'un type effectivement connu au compile-time, ou le fait que le comportement rajouté dans l'interposition soit relativement générique (c.a.d s'applique à des ensembles de méthodes).

On pourrait, par exemple, faire de l'interposition dynamique pour faire des appels de méthodes sur des objets distants implantant une certaine interface (une sorte de R.M.I. simplifié sans Stub), ou encore pour interposer des controles avant et après tous les appels de méthodes destinés à un objet (vérification d'invariant de classe, mise en place de transactions,...).

Schéma simplifié d'utilisation pour des appels distants:

Une réalisation correcte étant très complexe nous allons esquisser un mécanisme de mise en place d'appels distants :

```
public interface RemoteService extends Remote {
   Clef remoteGet(Object support, DescripteurMethode factory,
                  Object[] arguments) throws RemoteException ;
   Object remoteInvoke(Clef support, DescripteurMethode meth,
                  Object[] arguments) throws RemoteException ;
}
```

et maintenant un mécanisme rendant un objet dynamique et basé sur les mécanismes définis par `java.lang.reflect.Proxy` :

```
public class RemoteFacts {
   private RemoteService server ;

   private class Invoker implements InvocationHandler {
      Clef key ;
      public Object invoke(Object proxy, Method method,
                           Object[] args) throws Throwable {
            // le service demandé sur l'objet
            // est repassé au serveur
            //pour simplifier on n'a pas tenu compte des exceptions
           Object res =  server.remoteInvoke(key,
                  new DescripteurMethode(method, key), args) ;
           return res ;
      }

      Invoker(Clef key) {
         this.key = key ;
      }
   }
```

```
// sert à générer un objet sur le serveur en spécifiant
// une méthode “factory” sur un objet
// le serveur renvoie une “clef” qui lui permettra
// de retrouver l’objet effectif

   public Object remoteObjectAs(Class interf,
       Object factorySupport, Method factory,Object[] args)
              throws Exception{

        Clef key = server.remoteGet( factorySupport,
               new DescripteurMethode(factory), args) ;
         // on génère le Stub dynamique!
         // (manque les exceptions,etc....
        return Proxy.newProxyInstance (interf.getClassLoader(),
                new Class[] {interf}, new Invoker(key)) ;
   }
...
```

Ici l’objet généré par `newProxyInstance` sera conforme au contrat défini par l’interface demandée, et déléguera des appels de méthodes à un object correspondant connu uniquement sur le serveur. Dans l’exemple c’est l’instance de `Invoker` qui est chargé d’opérer la délégation.

Utilisation :

```
// on ne précise pas ici la nature de l’objet “banque”
// ni le fonctionnement des Classes “Clef” et
// “DescripteurMethode”

CompteDistant cpt = (CompteDistant) rmFact.remoteObjectAs(
            CompteDistant.class,banque, getCompteMeth,
               new Object[]{nomClient});

// et maintenant on peut appeler les méthodes de CompteDistant
// sur l’objet résultant : c’est un objet de plein droit
// qui implante l’interface demandée
```

Notion de "Bean"

Le développement basé sur des composants ne se limite pas à deux phases : développement / génération de code et exécution. D'autres phases intermédiaires sont possibles puisque les codes développés dans la première phase peuvent être réutilisés pour d'autres développements, assemblés dans le cadre de déploiement, etc.

Les capacités de réflexion / introspection de Java permettent d'analyser un code binaire existant pour rechercher certaines propriétés qui pourront être exploitées dans ces phases intermédiaires d'assemblage de composants. Si entre le producteur "primaire" de code et l'utilisateur de composant on met en place des conventions sur ces propriétés des codes on pourra ainsi faciliter l'action d'outils permettant d'assembler des composants, de générer des codes de déploiement, etc.

On appelle ces conventions des conventions de `Bean`. La convention de `Bean` initiale a surtout été orienté vers l'utilisation de composants graphiques mais il existe maintenant d'autres conventions complémentaires comme `Entreprise Java Bean` (composants d'entreprise déployés dans un serveur d'application) ou `Federated Management Bean` (composants de services d'administration dynamique sur le réseau -voir JIRO: une extension de JINI consacrée à l'administration répartie-).

Les packages `java.beans` *et* `java.beans.beancontext`

Ces packages fournissent les interfaces, les classes fondamentales pour exploiter la convention `Bean` initiale. Quelques points :

- Dans un objet *bean* sont nommés “propriété” des attributs “particulièrement visibles”, leur modification peut éventuellement permettre de paramétrer la mise en oeuvre de l’objet. Par convention on a une propriété si on a une paire de méthodes accesseur/mutateur de la forme `setXXX(typeX)` et `typeX getXXX()` (XXX devient le nom de la propriété)

- On peut mettre en place un système de gestion d’événements liés à la modification de propriétés. Certains *bean*s pourront par exemple apporter leur veto à des modifications sur un autre *bean*.

- Un objet `Introspector` permet de consulter de manières souple les caractéristiques *bean* d’un objet. Il opère par reflexion/introspection classique mais aussi en recherchant un objet `BeanInfo` qui a pour rôle de documenter un autre objet. La fourniture d’un tel objet d’information permet de modifier les résultats de l’introspection classique (par exemple en filtrant et en limitant l’exposition de certaines méthodes), mais aussi en fournissant des informations complémentaires qui sont susceptibles d’être exploité par programme (ou de manière “lisible” par un être humain).

- On peut mettre en place des mécanismes simplifiés de persistance pour sauvegarder une instance . La restauration d’un *bean* depuis un “bean confit” est un moyen de créer une instance comportant un état déja (partiellement) paramétré.

- Les services de `beancontext` permettent d’organiser le déploiement dynamique de *beans* au sein d’un “conteneur” qui leur fournit des informations sur les services disponibles dans le contexte et sur les événements d’ajout et de suppression d’autres *beans* au sein du conteneur.

XMLEncoder, XMLDecoder

Ces deux classes du package `java.beans` sont à la fois une application typique du mécanisme de *beans* et une partie intégrante des systèmes de persistance / échange de composants.

Les objets sont mis dans un format qui s'appuie sur les connaissances de l'API publique et ceci permet de ne pas tenir compte d'évolutions de détails de l'implantation effective du code (au contraire de la linéarisation classique). De plus cette sauvegarde se fait en XML ce qui rend ce format très pratique pour générer des tests (que l'on peut écrire ou modifier "à la main").

On peut sauvegarder en XML des objets qui ne sont pas des *beans* classiques mais pour lesquels le processus de reconstitution est documenté dans une meta-classe `BeanInfo` :

```
public class Client {
   public final int id ;
   public final String nom ;
   private String adresse ;
   private long nimporteQuoi ;

   public Client (int id, String nom, String adr) {
      this.id = id ;
      this.nom = nom ;
      adresse = adr ;
   }

   public String getAdresse() {
      return adresse  ;
   }

   public void setAdresse(String adr) {
      adresse = adr ;
   }
         ....
}
```

Le `BeanInfo` (à titre d'exemple: le processus complet n'est pas ici explicité)

```
public class ClientBeanInfo extends SimpleBeanInfo{
   private PersistenceDelegate persD =
        new DefaultPersistenceDelegate(
                    new String[] {  "id" , "nom", "adresse"});

      public BeanDescriptor getBeanDescriptor() {
         return new BeanDescriptor(Client.class) {
            {setValue("persistenceDelegate", persD);}
         } ;
      }
}
```

Un programme de sauvegarde

```
Client cli1 = new Client(133, "dupond" , "rue des lois") ;
XMLEncoder enc = new XMLEncoder(new FileOutputStream(nomFic)) ;
enc.writeObject(cli1) ;
...
```

Le contenu du fichier XML :

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.0" class="java.beans.XMLDecoder">
 <object class="dyn.Client">
  <int>133</int>
  <string>dupond</string>
  <string>rue des lois</string>
 </object>
</java>
```

En lecture par `XMLDecoder` l'objet sera reconstitué par appel du constructeur avec les paramètres corrects (issus de la sauvegarde des attributs significatifs).

Pour plus de détails lire :
`http://java.sun.com/products/jfc/tsc/articles/persistence4`

Références faibles 16

Points essentiels

Les techniques dites de “références faibles” permettent d’interagir avec le glaneur de mémoire (*Garbage Collector*) pour lui permettre de recupérer certaines références d’objets non-critiques, ou pour être averti si certains objets sont désalloués en mémoire (ou en passe de l’être).

Les références `java.lang.ref.Reference` sont des objets qui encapsulent une autre référence qui est gérée d’une manière particulière.

Seront abordés les `Reference`s de type :

- `SoftReference`
- `WeakReference` (ainsi que `WeakHashMap`)
- `PhantomReference`

Dans ce chapitre nous utiliserons le terme générique “référence faible” pour recouvrir ces différentes catégories.

Problématiques de caches mémoire

Un des caractéristiques de Java est que la désallocaiton des objets en mémoire n'est pas sous le contrôle du programmeur. Une tâche de fond de basse priorité le glaneur de mémoire *(Garbage Collector*) s'occupe de rechercher les objets qui ne sont plus référencés, désalloue leur empreinte dans le tas et "tasse" la mémoire pour mieux la gérer. Pour permettre des évolutions des algorithmes de nettoyage mémoire la norme ne fixe que très peu de contraintes sur le comportement des *garbage-collector*s. Le programmeur sait qu'il peut "aider" une meilleure gestion de la mémoire en mettant explicitement certaines références à `null` et après certaines opérations de "nettoyage" il peut explicitement faire appel à `System.gc()` (dans ce dernier cas il ne lui est même pas garanti que le *garbage-collector* répondra à sa requête!)

Or il existe des situations où un certain degré de collaboration avec le glaneur de mémoire est souhaitable. Un de ces cas est lorsque l'on souhaite gérer des caches mémoire c'est à dire que pour des raisons d'optimisation on cherche à conserver des objets en mémoire mais de manière raisonnée et compatible avec les autres exigences de demande de place. Deux exemples:

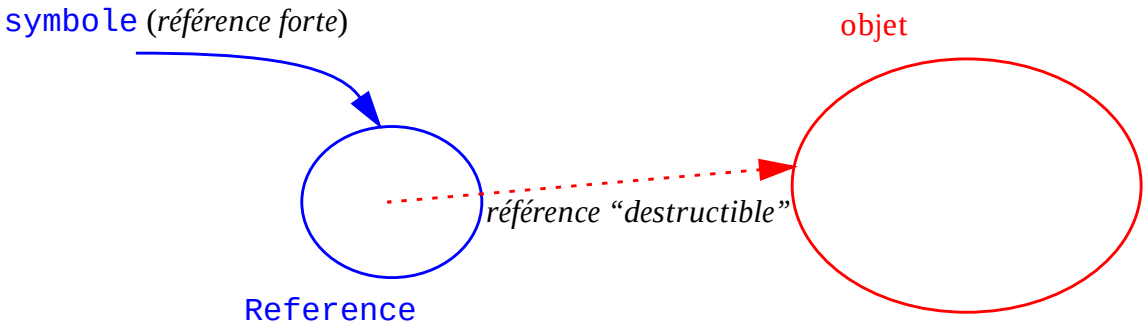
- Soit un programme qui affiche des images (représentation en mémoire couteuse); selon les interactions de l'utilisateur certaines images sont affichées souvent, d'autres non. Pour optimiser la gestion de la mémoire on souhaite adopter le comportement suivant: lorsqu'une image est demandée elle est chargée depuis une URL (opérations couteuse en temps!), les images sont conservées en mémoire mais lorsqu'on a besoin de place les images les moins récemment accédées seront eliminées: si l'utilisateur re-demande une image éliminée elle sera rechargée depuis son URL d'origine.

- Soit un programme qui réalise un calcul très couteux en temps et en ressources. Des statistiques font apparaître qu'il n'est pas rare que ce calcul soit appelé avec les mêmes paramètres. On décide alors de mémoriser les résultats des précédents calculs: si le calcul a déjà été effectué on sera en mesure de rendre directment le résultat sans opérer un nouveau calcul couteux. La mémoire n'étant pas indéfiniment extensible à un moment donné il faudra se résoudre à abandonner tout ou partie des "stocks" de résultats.

Les objets références

La package java.lang.ref introduit des objets `Reference`s. Ce sont des objets qui encapsulent une reférence vers un autre objet.

L'idée est que le programmeur puisse conserver une "référence forte" vers un objet `Reference`, mais la référence contenue (*referent object*) fait l'objet d'une convention avec le *garbage-collector*. Ce dernier peut éventuellement récupérer l'objet contenu qui devient inaccessible au programmeur.



L'accès indirect à l'objet cible se fait par la méthode get() :

```
monObjet = (TypeMOnObjet) maReference.get() ;
if(monObjet == null) { // ah tiens! GC est passé!
```

Bien entendu on ne doit pas conserver de "référence forte" sur l'objet cible pointé par la référence faible (l'objet ne serait pas récupéré par le glaneur et la stratégie serait mise en echec).

Il existe différent types standard de "références faibles" en fonction de la stratégie attendue du glaneur de mémoire.

Il est possible d'associer à un objet `Reference` une file `ReferenceQueue` dans laquelle le glaneur mettra l'objet `Reference` après avoir recupéré l'objet cible. De cette manière le glaneur décide de signaler au programme son action sur la `Reference` : on peut donc mettre en place une surveillance de cette file (bloquante: méthode `remove()`, ou non bloquante: méthode `poll()`) et décider d'actions correspondantes (par exemple détruire la "référence forte" sur l'objet `Reference` lui-même).

SoftReference

Voici, par exemple, le code simplifié d'une Applet qui met en place la stratégie de cache d'images exposée précédemment.

```
import java.applet.* ;
import java.awt.* ;
import java.net.* ;
import java.lang.ref.* ;

public class WeakApplet extends Applet {
   SoftReference ref ;
   URL base ;
   String nom ;

   public void init() {
      base = getDocumentBase() ;
      nom = getParameter("image");//se proteger contre erreur ici!
      ref = new SoftReference(getImage(base,nom)) ;
   }

   public void paint(Graphics gr) {
      Image img = null ;
      if ( null == (img = (Image) ref.get())) {
         img = getImage(base, nom) ;
         ref = new SoftReference(img) ;
      }
      gr.drawImage(img, 50, 50, this) ;
   }
}
```

SoftReference

Les objets cibles référencés au travers de références faibles de type `SoftReference` sont récupérées à la discrétion du glaneur de mémoire. Celui-ci est toutefois encouragé à essayer de les éliminer en épargnant en priorité les objets les plus récemment créés ou accédés. En ce sens ces références sont “moins faibles” que les objets `WeakReference`.

On utlisera des `SoftReference`s pour suivre des stratégies basées sur l’historique des accès et d’une manière analogue on pourra utiliser des `WeakReference`s pour avoir des stratégies moins discriminantes (et probablement plus performantes).

On notera, dans le code d’illustration, qu’une fois qu’on a constaté que l’objet cible a disparu, il faut créer une nouvelle référence pour contenir un objet reconstitué.

WeakReference

Les références `WeakReference` contiennent des objets cibles qui sont récupérées librement par le glaneur de mémoire.

Souvent ces objets servent de “représentant canonique” pour un autre objet c’est à dire de clef caractéristique par laquelle on peut, potentiellement, retrouver un objet donné. Pour pousser cette analogie nous alllons reprendre l’exemple du cache de calcul exposé précédemment et utiliser comme table de recherche un objet de type `java.util.WeakHashMap`.

L’objet caractéristique d’un calcul est sa combinaison de paramètres. Dans notre exemple le calcul “couteux” prendra deux paramètres entiers et nous allons décrire un objet représentant cette paire:

```
public class Parm2 {
   int x,y  ;

   public Parm2 (int a, int b) {
      x = a; y = b ;
   }

   public boolean equals(Object autre) {
      Parm2 oth = (Parm2) autre ;
      return (x == oth.x) && ( y == oth.y) ;
   }

   public int hashCode() {
      return x ^ y ;
   }
}
```

On notera que, cet objet servant de clef pour une table de *hash*, il faut soigneusement définir la combinaison des méthodes `equals()` et `hashCode()`.

Dans la `WeakHashMap` les clefs sont automatiquement générées comme références faibles et doivent être le seul point d’accès possibles pour les valeurs quelles représentent. Le glaneur de mémoire va pouvoir recupérer ces valeurs selon ses besoins.

```
import java.util.* ;
import java.lang.ref.* ;

public class CalculEnStock {
   WeakHashMap dict = new WeakHashMap() ;

   Long grosCalcul(int x, int y) {
         .....// gros calcul: (si vous n´êtes pas convaincu
              // essayez la fonction d'ackermann!)
      return new Long(res) ;
   }

   long calculStocké(int x, int y) {
      Parm2 parms = new Parm2(x,y) ;
      Long res = (Long) dict.get(parms) ;
      if( res == null) {
         res = grosCalcul(x,y) ;
         dict.put(parms, res) ;
      }
      return res.longValue() ;
   }
```

Ici le dictionnaire des résultats grossit au fur et à mesure des besoins, mais le glaneur a la faculté de mettre le holà à cette croissance et d'éliminer des résultats.

Cet exemple est de portée pédagogique : l'élimination des résultats stockés est radicale et peu discriminante (Il n'y a pas ici de stratégie particulière pour choisir les candidats à l'élimination)[1].

1. Une réalisation plus satisfaisante pourrait par ex. faire appel à `java.util.LinkedHashMap` utilisant un paramètre spécifiant un ordre basé sur l'accès et une taille limite controlée par `removeEldestEntry`

Opérations liées à l'abandon d'un objet, finaliseurs, `PhantomReference`s

Avant de récupérer l'empreinte mémoire d'un objet le garbage-collector appelle sa méthode `finalize()` (qui doit être redéfinie si on veut un comportment associé à cette phase de la vie de l'objet).

L'utilisation de `finalize()` est délicate : l'appel survient à des moments imprévus, peut être exécuté dans des *thread*s différents et il n'est pas garanti que tous les objets aient été récupérés au moment de l'arrêt de la machine virtuelle (la méthode `System.runFinalizersOnExit()` a été rendue obsolete du fait de ces risques).

Les `PhantomReference`s permettent d'associer, de manière plus souple, des méthodes à cette phase de fin de vie .

Exemple. Soit la définition d'une interface:

```
public interface DerniereVolonte {
   public void adieu() ;
}
```

et maintenant la définition

```
import java.lang.ref.* ;

public class PresqueFantome extends PhantomReference {

   private DerniereVolonte vol ;

   public PresqueFantome(
        DerniereVolonte vl, Object obj, ReferenceQueue queue){
      super(obj,queue) ;
      vol = vl ;
   }

   public void preMortem() {
      vol.adieu() ; // se proteger des exceptions
      clear() ;
   }
}
```

Les PhantomReferences sont obligatoirement associées à une ReferenceQueue. Quand le glaneur de mémoire va recupérer l'objet cible de la reférence, son finaliseur est appelé et la référence est mise dans la file. A partir de ce moment, bien que l'empreinte de l'objet n'aie pas été recupérée, il est impossible d'obtenir l'objet (de toute façon `get()` rend toujours `null` sur une telle référence) - par contre au cours du traitement de la reférence il faudra explicitement annuler la référence vers l'objet cible (`clear()`).

Utilisations possibles de nos reférence PresqueFantomes: admettons qu'elles aient été toutes construites avec la file suivante :

```
final ReferenceQueue ref = new ReferenceQueue() ;
```

Recupération par un *thread* en cours d'exécution:

```
Thread th = new Thread("fantomes") {
         {setDaemon(true) ;}
         public void run() {
            while(true) {
               try {
                 Reference curRef = ref.remove();// bloquant!
                 ((PresqueFantome)curRef).preMortem() ;
                  // suite du ménage .....
               } catch(Exception exc) {
                  curlogger.warning(exc.getMessage()) ;
               }
            }
         }
      } ;

       th.start() ;
```

Notons que dans dans ce cas on ne déclenche la méthode associée au décès que si le glaneur est passé pour l'objet cible. Il peut être utile de prévoir de faire quelque chose pour les autres.

Point intermédiaire : "finalisation"

Les opérations déclenchées au moment du passage du *garbage-collector* et les opérations de fermeture de l'application (*shutdown hook*) sont particulièrement sensibles car on ne contrôle pas complètement l'état de l'application au moment où elles se déclenchent. Il faut particulièrement veiller aux déclenchements intempestifs d'exceptions et aux problèmes de concurrence d'accès.

exercice :

- En réutilisant l'interface `DerniereVolonte` de l'exemple précédent écrire une classe `Notariat` disposant des méthodes de classe :

```
public static Reference fabriqueVoeu(Object o, DerniereVolonte v)

public static void executeVoeu(Reference r)
```

Telles que : si un objet enregistre ses dernières volontés celles-ci sont obligatoirement exécutées:

- soit par appel explicite de `executeVoeu`
- soit au moment de la recupération de l'objet par le glaneur de mémoire .
- soit à la fin de l'application (si, par exemple, on a conservé jusqu'au bout une référence forte sur l'objet, ou si le glaneur n'est pas passé).Récupération au moment de l'arrêt de la machine virtuelle: Exemple:

```
Runtime.getRuntime().addShutdownHook(new Thread("notaire") {
        public void run() {
                // poll() sur la file de references
                // + executer voeu sur refs survivantes
        }
    }) ;
```

Discussion: quelles utilisations pratiques verriez-vous à ce mécanisme?
Par ailleurs si vous réalisiez de cette manière un code fondamental susceptible de nombreuses ré-utilisations dans le temps comment le rendriez-vous paramétrable ou redéfinissable?

Compléments

Autres techniques d'abandon d'objets

Il existe d'autres techniques pour décider d'abandon d'objets - particulièrement sur des serveurs -. Ainsi les techniques de "bail" (*lease*) permettent d'accorder à un client un contexte de service de durée limitée.

Du point de vue du code client cela veut dire que le contexte sur le serveur n'est pas forcément garanti et qu'il faut "renouveler le bail" régulièrement. Dans certaines stratégies le bail n'est même pas garanti pour une durée donnée (voir service de "lease" sur JINI). Effectivement dans une situation qui devient tendue un serveur peut décider de raccourcir les délais. On peut donc facilement imaginer que la réalisation de stratégies fasse appel à des combinaisons de techniques qui tiennent à la fois compte de la qualité du service et des contraintes technique du serveur. Ces techniques de syntonisation peuvent devenir très complexes et il peut devenir intéressant de les déléguer à des outils spécifiques.

Utilisation de code natif : J.N.I 17

Points essentiels

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 complétement écrit en JAVA :

- Une grande quantité de code compilé existe déja 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 adapté à une partie critique de l'application 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 */

#ifndef _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'équivallent 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 collecto*r).

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 déréférencements 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 recupérer 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'accesseur `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 `[`) 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 recupérer 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

Pour un exemple développé voir dans le répertoire `src/launcher` de l'installation du SDK.

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

Java et le Web: les applets 18

Points essentiels

- Les Applets constituents 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 la lie au navigateur qui la met en oeuvre (cycle de vie de l'Applet).

- Les codes qui s'exécutent au sein d'une Applet sont soumis à des restrictions de sécurité.

Applets

Une *Appliquette* (*Applet*) est une portion de code Java qui s'exécute dans l'environnement d'un navigateur au sein d'une page HTML. Elle diffère d'une application par son mode de lancement et son contexte d'exécution.

Une application autonome est associée au lancement d'un processus JVM qui invoque la méthode `main` d'une classe de démarrage.

Une JVM associée à un navigateur peut gérer plusieurs Applets, gérer leur contexte (éventuellement différents) et gérer leur "cycle de vie" (initialisation, phases d'activité, fin de vie).

Lancement d'une Applet

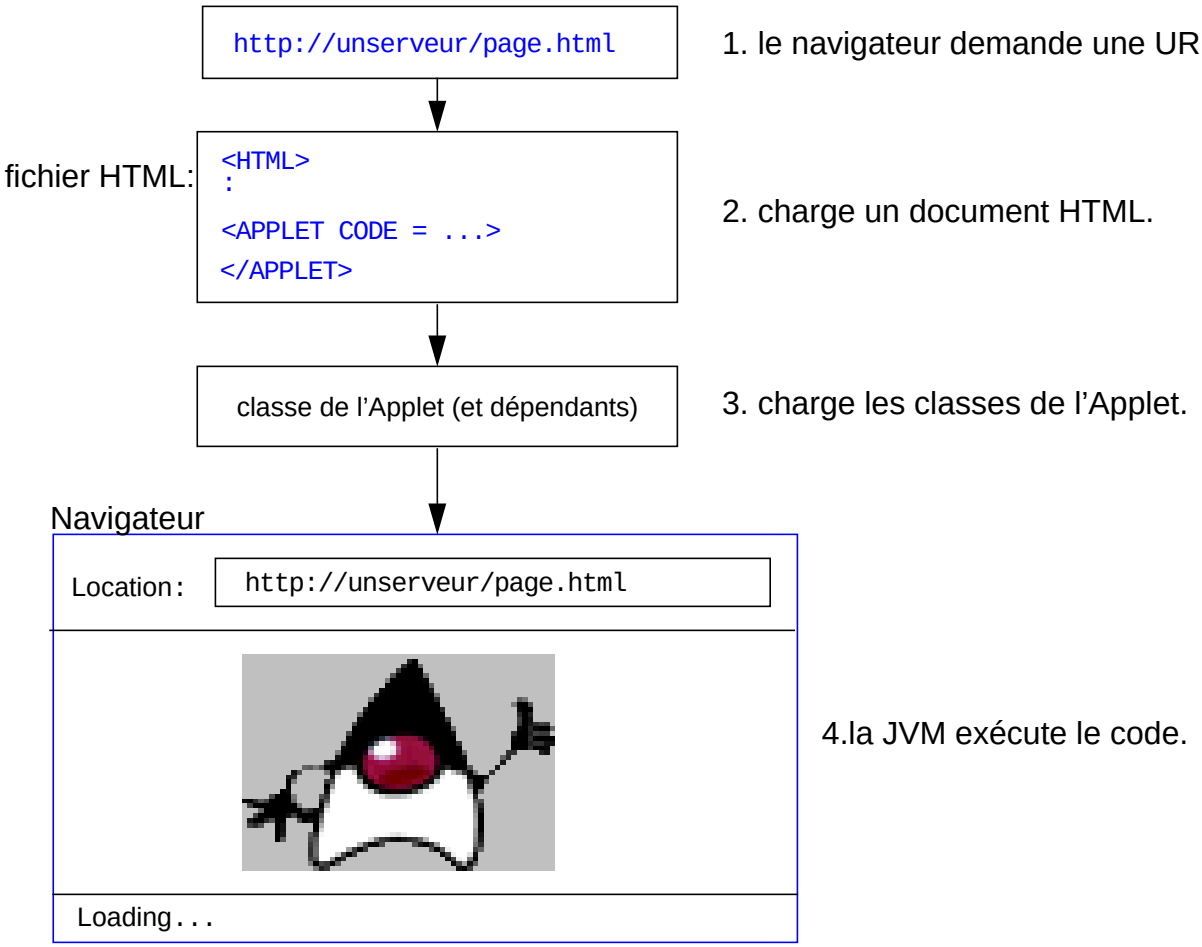
Une applet s'exécutant dans le cadre d'une "page" HTML, la requête de lancement et les paramètres associés sont en fait contenus dans le fichier source décrivant la page.

Le langage HTML (voir `http://www.w3.org/MarkUp`) permet de définir une présentation à partir d'un texte contenant des *balises* (qui sont des instructions de mise en page). Certaines de ces balises demandent le chargement de données non-textuelles comme des images ou des Applets Java.

Le lancement d'une Applet suppose donc :

- La désignation d'un document au navigateur au travers d'une adresse URL (voir `http://www.w3.org/Adressing`)
- Le chargement et la mise en page du document HTML associé
- Le chargement du code de l'Applet spécifié dans le document (et éventuellement le chargement des codes des classes distantes appelées par l'Applet).
- La gestion, par la JVM du navigateur, du cycle de vie de l'Applet au sein du document mis en page.

Applets: lancement



Exemple d'utilisation de la balise APPLET :

```
<P> et maintenant notre Applet :

<APPLET code="fr.gibis.applets.MonApplet.class"
        width=100 height=100>
</APPLET>
```

Applets: restrictions de sécurité

Dans ce contexte on va exécuter des codes de classe : classses “distantes” et classes “système” de la librairie java locale. Pour éviter qu’un code distant puisse réaliser des opérations contraires à une politique de sécurité élémentaire, des règles par défaut s’appliquent (*sandbox security policy*):

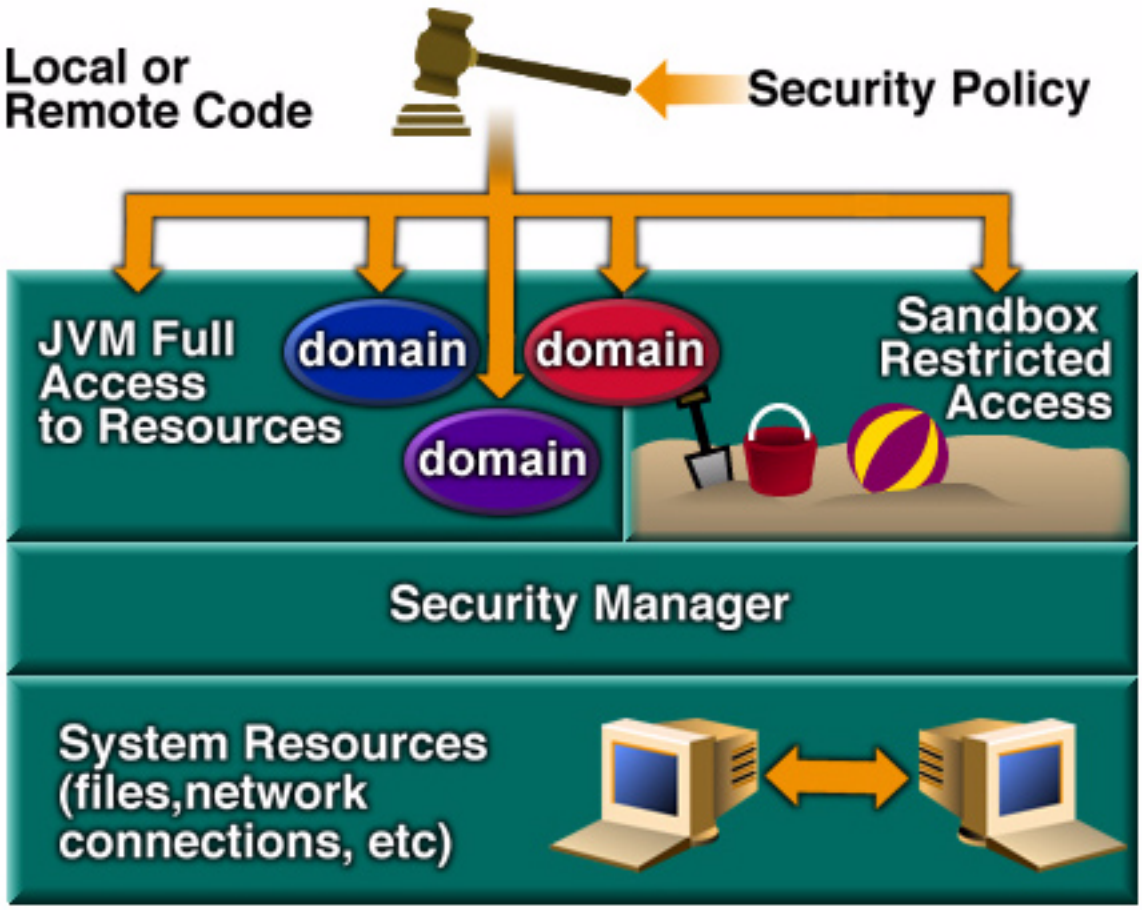
- L’Applet ne peut obtenir des informations sur le système courant (hormis quelques informations élémentaires comme la nature du système d’exploitation, le type de J.V.M., etc.).
- L’Applet ne peut connaître le système de fichiers local (et donc ne peut réaliser d’entrée/sortie sur des fichiers).
- L’Applet ne peut pas lancer de processus
- Les communications sur réseau (par *socket*) ne peuvent être établies qu’avec le site d’origine du code de l’Applet.

Bien entendu un code d’Applet ne peut pas contenir de code “natif” (par essence non portable).

A partir de la version 2 de Java un système standard de mise en place de domaines de sécurité (*Protection Domain*) permet d’assouplir ces règles pour permettre certaines opérations à des codes dûment authentifiés venus de sites connus. On peut ainsi imaginer que l’Applet qui vous permet de consulter votre compte en banque vous permet aussi de rapatrier des données à un endroit du système de fichier défini par vous.

Applets: restrictions de sécurité

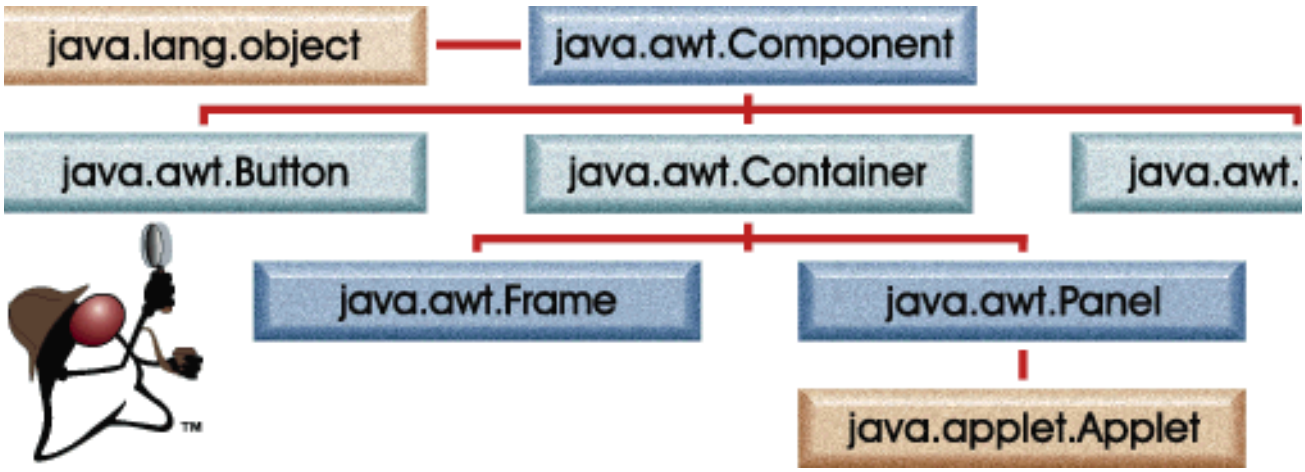
Le modèle de sécurité Java 2 s'applique dès qu'un `SecurityManager` est présent et ce aussi bien pour du code téléchargé que pour du code local



Hiérarchie de la classe Applet

De par l'incorporation d'une Applet à une présentation graphique (la "page" du navigateur) toute Applet est, de manière inhérente, une classe graphique (même si on peut créer des Applets qui n'ont aucune action de nature graphique). A la base l'incorporation d'une Applet dans une page HTML revient à signaler au navigateur qu'une zone de dimensions données n'est plus gérée directement par l'interpréteur de HTML mais par un programme autonome qui prend le contrôle de la représentation graphique de cette zone.

Toute classe qui est une Applet doit dériver de `java.applet.Applet` qui est elle-même une classe dérivée de `java.awt.Panel` qui représente un "panneau" graphique dans la librairie graphique AWT.



Applets: groupes de méthodes

Une Applet étant un `Container` graphique, elle hérite des méthodes qui permettent de disposer des composants graphiques dans la zone gérée par la classe. Ce point est traité du point de vue de la programmation AWT “classique”, et, d’un autre coté , nous allons nous attacher à trois groupes de méthodes:

- Méthodes qui sont appelées par le système graphique pour la notification d’une demande de rafraîchissement d’une zone de l’écran:
  `repaint(), update(graphics), paint(Graphics)`
  Dans le cas où l’Applet souhaite gérer le graphique de bas niveau (au lieu de laisser agir le système automatique attachés à des composant AWT de haut niveau) elle doit conserver un modèle logique des éléments graphiques qu’elle gère et être capable de les redessiner à la demande.

- Méthodes spécifiques aux Applets et qui leur permettent de demander au navigateur des informations ou des actions liées au contexte: informations sur la page HTML courante, chargement d’une ressource sur le site d’origine de la page, etc.

- Méthodes spécifiques aux Applets et à leur cycle de vie : le navigateur doit pouvoir notifier à l’Applet qu’il veut l’initialiser (`init()`), qu’il veut la rendre active ou inactive (`start(), stop()`), ou qu’il veut la “détruire” (`destroy()`).
  Par défaut ces méthodes ne font rien et il faut en redéfinir certaines d’entre elles pour obtenir un comportement significatif de l’Applet.

H.T.M.L.: la balise `Applet`

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=valeur>]
  [<PARAM name=Attribut2 value=valeur>]
  . . . . .
  [HTMLalternatif]
</APPLET>
```

Exemple :

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

Evolutions (HTML 4) :

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

HTML: la balise `Applet`

. La balise HTML APPLET comporte les attributs suivants :

- 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. Ce point ne sera pas abordé dans ce cours.

- object=*objectFile.ser* permet de spécifier une instance d'objet à charger. Ce point ne sera pas abordé dans ce cours.

`<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()`.

Méthodes du système graphique de bas niveau

Applet héritant de Panel il est possible de modifier l'aspect en utilisant les méthodes graphiques des composants.

Voici un exemple minimal d'Applet qui écrit du texte de manière graphique:

```
import java.awt.* ;
import java.applet.Applet ;

public class HelloWorld extends Applet {

        public void paint(Graphics gr) {
                gr.drawString("Hello World?", 25 ,25) ;
        }
}
```

Les arguments numériques de la méthode drawString() sont les coordonnées x et y du début du texte. Ces coordonnées font référence à la "ligne de base" de la police. Mettre la coordonnée y à zero aurait fait disparaître la majeure partie du texte en haut de l'affichage (à l'exception des parties descendantes des lettres comme "p", "q" etc.)

Méthodes d'accès aux ressources de l'environnement

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 cheminoms relatifs (par ex: "../../images/truc.gif"). Attention toutefois : certaines configurations 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 ressources 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`.

Méthodes d'accès aux ressources de l'environnement

Utilisation de la récupération de paramètres. Exemple de source HTML:

```
<APPLET code="Dessin.class" width=200 height=200>
        <PARAM name="image" value="duke.gif">
</APPLET>
```

Le code Java correspondant :

```
import java.awt.*;
import java.applet.Applet;

public class Dessin extends Applet {
        Image img ;

        public void init() {// redef. méthode standard cycle vie
                String nomImage = getParameter("image") ;
                if ( null != nomImage) {
                        img = getImage(getDocumentBase(),
                                nomImage);
                }
        }

        public void paint (Graphics gr) {
                if( img != null) {
                        gr.drawImage(img,50,50, this) ;
                } else {
                        gr.drawString("image non chargée",
                                25,25) ;
                }
        }
}
```

Les méthodes de chargement de media comme `getImage()` sont des méthodes *asynchrones*. On revient de l'appel de la méthode alors que le chargement est en cours (et, possiblement, non terminé). Il est possible que `paint()` soit appelé plusieurs fois au fur et à mesure que l'image devient complètement disponible.

Méthodes du 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 la fenêtre courante du navigateur est restaurée après avoir été iconisée ou lorsque la page qui héberge l'applet 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 d'être en phase active. 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.

Méthodes du cycle de vie

```
// Suppose l'existence du fichier son "cuckoo.au"
// dans le répertoire "sounds" situé dans
// le 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");
                // attention syntaxe d'URL!!!
        }

        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();
        }
}
```

Rappels: syntaxe, spécifications simplifiées 19

Contenu:

Rappels sur la syntaxe (simplifiée)

Condensé des spécifications d'utilisation des variables, méthodes, constructeurs.

Pour des descriptions plus complètes voir le document de spécification du langage (J.L.S.) voir:
`http://java.sun.com/docs/books/jls/`

Plan général d'une classe

Plan général des déclarations de classe premier niveau:

```
package yyy;
import z.UneCLasse ; // A préférer
import w.* ;
public class XX extends YY implements AA, BB {
        // public ou "visibilité package"
                  // une classe publique par fichier
        // si méthode "abstract" la classe doit etre "abstract"
        // ordre indifférent sauf pour les blocs et expressions
        // d'initialisation de premier niveau

        //MEMBRES STATIQUES
        variables statiques
                  var. statiques initialisées
                  var. statiques "final"/constantes de classe
        méthodes statiques
        classes ou interfaces statiques
                  classes statiques avec accès privilégié
                            avec la classe englobante.

        //MEMBRES D'INSTANCE
        variables d'instance
                  var.d'instance initialisées
                  var d'instance "blank final"
        méthodes d'instance
                  "patrons" de méthodes d'instance (abstract,...)
        classes d'instance
                  classes avec accès instance englobante

        //BLOCS DE PREMIER NIVEAU
        blocs statiques
         /* évalués au moment du chargement de la classe */
        blocs d'instance
         /* évalués au moment de la création de l'instance */

        // CONSTRUCTEURS
        constructeurs
}
```

Plan général d'une interface

```
public interface JJ extends BB, CC {
        var. statiques "final"/constantes de classe
        classes ou interfaces statiques
        "patrons" de méthodes publiques
}
```

Rappels syntaxiques (simplifiés)

Nota: pour une description complète en format B.N.F. voir les documents de spécification du langage (le formalisme adopté ici est très simplifié).

Déclarations de premier niveau

Déclarations de variable membre :

Descriptions simplifiées :

```
<modificateurs> <Type> <nom> ;
<modificateurs> <Type> <nom> = <expression> ;
```

Exemples :

```
private int x;
java.math.BigDecimal solde ;
public static final int MAX = 5 ;
```

exemples d'autres cas:

```
protected Truc[] tb ;
protected Truc[] tb2 = {new Truc(), new Truc()} ;
int x, y, z;
```

Déclarations de méthode membre

Descriptions simplifiées:

```
<modif.> <Type_res> <nom> (<liste type-nom>) <bloc methode>
<modif.> <Type_res> <nom> (<liste type-nom>)
                    throws <liste_exception> <bloc methode>
```

Exemples :

```
public BigDecimal getSolde() { return this.solde ;}

public final void deStocker( int nb) throws ExceptionStock {
          if (nb > this.stock) {
                    throw new ExceptionStock(this) ;
          }
          stock -= nb ;
}

public static void main(String[] args) {
          System.out.println(“test”) ;w
}
```

exemples d’autres cas :

```
int compareTo(Object autre) ; // dans une interface

public abstract void paint(Graphics gr) ; //abstrait

public MonObjet fabrique(final int num, String s) {
          .... // code
}
```

Si la méthode ne renvoie pas un type void, <bloc_méthode> doit contenir au moins une expression `return <expression du type demandé>;`

Déclarations de classes ou interfaces membres

Descriptions simplifiées:

```
<modif.> class <nom> <extends et implements decl.> <bloc classe>
<modif.> static interface <nom> <extends list> <bloc interface>
```

Exemples :

```
public static class Exception extends java.lang.Exception {
          double solde, dépassement;
          private Exception(String mess, double x, double y) {
                    super(mess) ;
                    solde =x ; dépassement = y ;
          }
}// dans classe “Compte” donne classe Compte.Exception

private class Accesseur implements Enumeration {
          ...// code de Accesseur
}

public static interface Taxeur {
          double getTaux(Sucrerie sucr) ;
}
```

Déclarations de constructeur

Descriptions simplifiées:

```
<modif.> <nom classe> (<liste type-nom>) <bloc construct.>
<modif.> <nom classe> (<liste type-nom>)
                    throws <liste_exception> <bloc construct.>
```

Exemples :

```
public Point( int x, int y) {
          this.x =x;
          this.y = y;
}

public SousTruc(String nom)throws IllegalArgumentException {
          super(nom) ;// le constructeur de Truc
                      // qui , lui meme peut declencher une exc.
}
```

Syntaxe simplifiée des blocs de code

Dans un bloc de code on peut trouver :

```
<declaration variable locale>
<déclaration de classe locale>
<instruction>
<structure de contrôle>
<rupture>
<bloc ou structure etiquetée>
```

déclaration variable locale

Descriptions simplifiées:

```
<Type> <nom> ;
<Type> <nom> = <expression> ;
```

Exemples:

```
double dx ;
double dy = 3.14 ;
```

Exemples d'autres cas:

```
final int ix = 7 ;
double dx, dy ;
int[][] tb = {{ 1,2,3} , {4,ix}} ;
```

instruction :

Les instructions pouvant prendre des formes détaillées très diverses une simple liste pour rappel:

```
<affectations>
<pre/post incrémentation/decrémentation>
<appel de méthode>
<invocation de constructeur>
<assertions> //java 1.4
```

structures de contrôle:

IF :

```
if (<expression booléenne>) <instruction ou bloc>
if (<expression bool>) <instr. ou bloc> else <instr. ou bloc>
```

Exemples:

```
if (estceVrai) {// préférer toujours avec accolades
          if( x == 0) return ;// ici encore lisible
          .....// code
} else {
          ...// code
}
```

SWITCH

```
switch (<expression entière>) {
          case <constante> : <liste instructions>
          case <constante> : <liste instructions> break;
          ....
          default: <liste instructions> [break] ;
}
```

Exemples:

```
switch(random.nextInt()%2) {
          case -1 : mess = "pas d'accord" ; break;
          case 1 : mess = "d'accord" ; break;
          default: mess = "ni pour, ni contre(bien au contraire!)";
}
```

WHILE:

```
while(<test_booléen>) <instruction ou bloc>
```

Exemples:

```
bonneRéponse = teste() ;
while(!bonneRéponse) {
          affiche("essaye encore!") ;
          bonneRéponse = reTeste() ;
}
```

DO...WHILE :

```
do <instruction ou bloc> while (<test booléen>) ;
```

Exemples:

```
do {
          bonneRéponse = teste() ;
          affiche(bonneRéponse?”bravo!”:”essaye encore!”) ;
} while (!bonneRéponse) ;
```

FOR:

```
for ( <initalisations> ; <test haut> ; <opérations bas de boucle>)
          <instruction ou bloc>
```

Exemples;

```
for(int ix = 0 ; ix <tb.length ; ix++) System.out.println(tb[ix]);

for(int ix=0, iy = tb.length;ix<tb.length; ix++, y--) {
          tb[ix] = iy ;
}

for(;;) {// forever
          sam.playItAgain() ;
}
```

TRY..CATCH..FINALLY

```
try {
        // code
}catch (<exception1> <nom>) {
        // code
}catch (<exception2> <nom2>) {
        // code
}....

try { //code
} catch (<exceptiom> <nom>) { // evt plusieurs catch
        // code
} finally { // code
}

try {
        // code avec déroutement ou propagation exception
}finally {
        // code garanti sauf arrêt machine virtuelle
}
```

déroutements:

RETURN:

```
return ; // dans méthode void ou constructeur
return <expression> ;// dans une méthode rendant un résultat
        // du même type que <expression>
```

BREAK:

```
break; // dans un bloc de code
break <étiquette> ;// dans bloc étiqueté
```

CONTINUE:

```
continue ; // dans une boucle
continue <étiquette> ; //saut bas de boucle étiquetée
```

THROW:

```
throw <throwable> ;
```

Variables

Deux aspects : déclaration , utilisation.

On ne peut utiliser une variable qui n'a pas été déclarée. Dans de nombreux cas on ne peut pas utiliser une variable qui n'a pas été initialisée (vérification de *compile-time*).

Critères à controler :

- Nom : un symbole (un nom) va désigner la variable. Il faut que ce nom ne soit pas un mot réservé, que ce "mot" ne constitue pas une gêne pour une variable déclarée avec le même "nom" (conflit de portée). Suivre les règles de nommage pour rendre votre code plus lisible.

- Type : toute déclaration d'un nouveau symbole doit s'accompagner d'une déclaration de type qui définit les droits et obligations liés à l'usage du nom. Attention les types scalaires primitifs et les types objets n'ont pas les mêmes comportements, ni les mêmes contraintes.

- Initialisation : à vérifier soigneusement. Le compilateur fait une vérification d'initialisation sur les variables automatiques (variables de bloc).

Forme générale des déclarations:

```
Type nomSymbole
        // exemples
        int ix              // un scalaire
        Integer iix         // un objet
        double[] tbd        // un tableau de scalaires
        UneClasse[] tbo     // un tableau d'objets
        UneClasse[][] tb2   // un tableau de tableaux
```

variables d'instance

Variables membres de l'instance. Chaque objet créé sur le modèle de la classe gère sa propre combinaison de valeurs associées à ces variables.

```
public class Compte {
        int numero ;
        BigDecimal solde ;
}
```

Les objets créés sur ce modèle: `compteDupond`, `compteDurand` ont des valeurs différentes pour `numero` ou `solde`.

- Déclaration : sous la forme `Type nom ;`
  (autres formes possibles : `Type nom1, nom2`, etc. et déclaration +initialisations)
  Ordre et emplacement des déclarations indifférent (sauf si initialisations)

- Modificateurs d'accès possibles: `public`, `private`, `protected`

- Initialisations : par défaut les variables d'instance sont initialisées à zéro au moment de la création de l'instance. Selon les types des variables cette initialisation peut avoir des signications différentes (*false* pour les `booleans`, *null* pour les références, etc.).
  En général il est conseillé de réaliser l'initialisation d'une variable d'instance soit dans un constructeur (valeur passée en paramètre du constructeur), soit dans une déclaration avec initialisation (par exemple pour l'allocation de structures de données):

```
public class Compte {
        int numero ;
        BigDecimal solde ;
        // "structure de donnée" initialisée
        Arraylist listOpérations = new ArrayList() ; \

        public Compte( int num, String depotInitial) {
                numero = num ;
                solde = new BigDecimal(depotInitial) ;
        }
}
```

Dans le cas où la variable d'instance est déclarée et initialisée l'ordre de déclaration des initialisations est important. Par ailleurs ces initialisations sont exécutées avant la partie du code du constructeur ne faisant pas appel à un autre constructeur (appel de `this(...)` ou `super(...)`).
Attention: aucune de ces opérations d'initialisation hors constructeur ne doit être susceptible de propager une exception controlée (voir également bloc d'initialisation d'instance).
Une variable d'instance peut être marquée `final` ("*blank final*"): dans ce cas elle doit être initialisée dans le constructeur (ou par une initialisation immédiate); dans un constructeur on ne peut initialiser une variable `final` héritée (puisque celle-ci a déjà été initialisée dans la construction d'une super-classe).

- Accès : on ne peut consulter une variable d'instance que si l'on dispose d'une instance :
  `compteDupond.numero ;`
  Dans le code de la classe on a implicitement une référence à l'instance courante (this) :

```
double getSolde() {// dans classe Compte
          return solde.doubleValue() ;
          // implicitement this.solde
}
```

Attention: ceci n'est pas vrai dans le code d'une méthode statique (comme main).
Autres cas : désignation d'une instance d'une classe englobante : `ClasseEnglobante.this.champ` et désignation d'un champ "caché" de la super-classe : `super.champ` (conseil: ne pas se mettre dans cette situation qui consiste à redéfinir un champ d'instance dans une sous-classe).

- Autres modificateurs : `transient, volatile`

variables de classe

Variables membres de la classe. Dans le contexte d'un ClassLoader toutes les instances ont accès à ces **variables partagées**.

```
public class Compte {
        static int compteurDInstances ; // partagé
        int numero ;
        BigDecimal solde ;
}
```

- Déclaration : sous la forme `Type nom ;`
  (autres formes possibles : `Type nom1, nom2`, etc. et déclaration +initialisations)
  Ordre et emplacement des déclarations indifférent (sauf si initialisations)

- Modificateurs d'accès possibles: `public, private, protected`

- initialisations : par défaut les variables de classe sont initialisées à zéro . Selon les types des variables cette initialisation peut avoir des signications différentes (*false* pour les `boolean`s, *null* pour les références, etc.).
  Pour réaliser l'initialisation d'une variable de classe il faut soit utiliser un bloc statique de premier niveau , soit une déclaration avec initialisation -ces initialisations sont effectuées au moment de l'initialisation de la classe (premier accès aux services de la classe après chargement par le ClassLoader)-:

```
public class Contexte {
        public static String osName =
                              System.getProperty("os.name");
        public static URL urlDoc ;
        static { // bloc statique
                  try { urlDoc = new URL("http://java.sun.com")
                  } catch(MalformedURLException exc){}
        }//ATTENTION si "urlDoc" est final : problèmes possibles
}
```

Les variables membres d'une interface sont implicitement `public static final`.

Dans le cas où la variable de classe est déclarée et initialisée l'ordre de déclaration des initialisations et des blocs statiques est important: les évaluations se font dans l'ordre au moment de l'initialisation de la classe.
Attention : les opérations d'initialisation ne peuvent faire appel à des méthodes d'instance de la classe courante (sans création d'une telle instance) et ne doivent pas être susceptibles de propager des exceptions controlées.

Une variable de classe peut être marqueé `final` et initialisée (ce qui ne veut pas nécessairement dire que ce sont des constantes évaluables au moment de la compilation).

- Accès : on peut accéder à une variable de classe de 3 manières:
  - en la qualifiant par le nom de la classe
    `MaClasse.maVariableDeClasse`
    `MaClasse.CONSTANTE_DE_CLASSE`
    ce sont les notations à préférer.
  - en la qualifiant par une référence de la classe
    `instanceDeMaClasse.maVariableDeClasse`
    (ou en utilisant implicitement ou explicitement `this`)
    Cette notation est à éviter car elle ne facilite pas la compréhension du code.
  - en la qualifiant par désignation de la super-classe :
    `super.variableDeClasse`
    ceci correspond à une situation de redéfinition d'une variable de classe dans une sous-classe (*data hiding*): à éviter
- Autre modificateur : `volatile`

variables de bloc

Il s'agit de variables temporaires: elles sont créées au moment où l'exécution atteint cette partie du code et détruites quand on sort de la portée du bloc courant (variables ou références allouées dans la pile).

- Déclarations :
  - instruction de déclaration dans un bloc

```
{
    int ix ;// n'importe où dans le bloc
    String s1, s2, s3 ;
    double dz = 3.14 ;
```

  - cas particuliers assimilés aux précédents :

```
for(int ix = 0, iy = tb.length; ix<tb.length; ix++, iy--) {
    // ix et iy n'existent que dans la portée du bloc "for"
```

  - paramètres

```
void methode(int ix, String nom) {
        // ix et nom en portée dans le bloc
....
Maclasse(String parm) { // paramètre constructeur
...
try {....
} catch (IOException exc){//exc en portée dans bloc catch
}
```

Attention: un symbole local dans un bloc peut "cacher" une variable d'instance ou une variable de classe, le compilateur ne le signale pas mais le code devient illisible. Par contre il ne peut pas porter le même nom qu'un symbole du bloc local ou d'un bloc englobant.

- Modificateurs :
  possibles : `final`,
  impossibles : `static`, `public`, `private`, `protected`, `volatile`, `transient` (attention: erreur de syntaxe difficile à comprendre, certains compilateurs donnent un message obscur!)

- Initialisations :

  - Les paramètres sont automatiquement initialisés (par l'appel de la méthode, ou la récupération d'une exception).
    La modification de la valeur d'un paramètre dans le bloc est sans effet sur le code appelant (voir cas particulier des "effets de bord").

  - Les autres variables de bloc dovient impérativement être initialisés avant d'être utilisées (condition vérifiée par le compilateur)

Méthodes

Deux aspects: définition d'un coté, utilisation de l'autre.

L'utilisation d'une méthode fait partie du contrat d'un type donné. Sauf dans le cas très particulier de l'exécution dynamique par introspection (package `java.lang.reflect`), le compilateur va s'efforcer de vérifier que l'appel est conforme à la définition (il consulte le fichier binaire ".class" pour cela).

Critères à controler :

- Nom : un symbole (un nom) va désigner la méthode. Plusieurs méthodes peuvent porter le même nom -avec des paramètres différents-(*surcharge*): il est préférable dans ce cas que l'intention sémantique soit la même.
  Pour éviter toute confusion ne pas utiliser le même nom qu'une variable ou qu'une classe (c'est possible!)
  Suivre les règles de nommage pour rendre votre code plus lisible.

- Signature : c'est la combinaison caractéristique d'une méthode comprenant le nom et la liste ordonnée des types des paramètres. Une classe ne peut pas avoir deux méthodes avec la même signature.

- Résultats : les méthodes de Java sont "fonctionnelles" par nature, elles prennent des paramètres en entrée et rendent un seul résultat pour l'exécution "normale". Si il n'y a pas de résultat on déclare un type `void`.
  Dès qu'une méthode déclare un résultat non-`void` on ne doit sortir "normallement" de la méthode que par l'appel d'un ou plusieurs `return` accompagné d'une valeur du type de retour demandé.
  Les résultats "anormaux" sont ceux qui découlent d'une erreur d'exécution: leur déclaration fait partie du "contrat" de la méthode.

```
Object[] tableauTrié(Object[] tb)
                    throws ClassCastException {
          Object[] res = (Object[]) Array.newInstance(
                    tb.getClass().getComponentType(), tb.length);
                    ...// copier dans res et trier
          return res ;
}
```

Méthodes d'instance

Méthodes membres de l'instance. Les instructions dans le corps de définition peuvent agir sur les variables propres à la méthode, les variables membres de l'instance courante, les variable de classe de la classe courante.

- Définition : sous la forme

```
TypeRésultat nom(liste_paramètres) throws liste_exceptions {
        //corps de définition
}
```

Les méthodes marquées `abstract` (et celles définies dans les interfaces) ou marquées `native` n'ont pas de corps de définition.

Des règles strictes gouvernent la déclaration d'une méthode comme redéfinition d'une méthode de la super-classe:

- on ne peut redéfinir une méthode de classe en méthode d'instance.
- on ne peut aggraver le "contrat" : rendre la méthode plus "privée" que la méthode originale, déclarer propager plus d'exceptions (ou des super-classes des exceptions déclarées).
- on ne peut modifier le type de résultat d'une méthode de même signature.

Nota: la définition d'un paramètre peut être d'une des formes:

```
Type nom
final Type nom
```

- Modificateurs :
  `public, private, protected, final, abstract`
  et `native, strictfp, synchronized`
  (`abstract` est incompatible avec `final` et `native`)
  Les en-têtes de méthodes de déclaration des interfaces sont implicitement `public abstract`.

- Appels : on ne peut appeler une méthode d'instance que si l'on dispose d'une instance (on "envoie un message" à l'instance):

```
monInstance.méthode(arg1, arg2, argn) ;
```

Dans le code de la classe on a implicitement une reférence à l'instance courante :

```
public class MaClasse {
        public void meth1() {
                ....
        }
        public void meth2() {
                ...
                meth1() ;
                // implicitement this.meth1()
        }
}
```

Autres cas :

- désignation d'une instance englobante :

```
ClasseEnglobante.this.methode() ;
//utile si methode est aussi definie dans classe locale
```

- désignation de la méthode de la super-classe (en cas de redéfinition locale)

```
super.toString() + "champ local:" + champlocal ;
// redefinition de toString() dans la classe fille
```

Les paramètres sont passés par valeur: c'est à dire qu'ils ont évalués, puis copiés dans la pile et c'est cette copie qui est récupérée par le code de la méthode. Les règles de promotion de type sont appliquées.

Méthodes de classe

Méthodes membres de la classe. Les instructions dans le corps de définition peuvent agir sur les variables propres à la méthode et les variable de classe de la classe courante. Elles ne peuvent agir sur des variables d'instance si ces variables ne sont pas qualifiées par rapport à une instance dûment créée.

- Définition : sous la forme

```
static TypeRes nom(liste_paramètres) throws liste_exceptions {
        //corps de définition
}
```

On peut redéfinir ainsi dans une classe une méthode statique de la super-classe (rupture du lien statique : *hiding*) -dispositif rare et peu conseillé-

- Modificateurs :
  `public, private, protected, final`
  et `native, strictfp, synchronized`.

- Appels : on peut appeler une méthode de classe:

  - en la qualifiant par le nom de la classe
    `MaClasse.maMéthodeDeClasse()`
    c'est la notation à préférer.

  - en la qualifiant par une référence de la classe
    `instanceDeMaClasse.maMéthodeDeClasse()`
    (ou en utilisant implicitement ou explicitement `this`)
    Cette notation est à éviter car elle ne facilite pas la compréhension du code (+ évaluation non polymorphique).

  - en la qualifiant par désignation de la super-classe :
    `super.maMéthodeDeClasse()`
    `((SuperClasse)instance).méthodeDeClasse()`
    ceci correspond à une situation de redéfinition d'une méthode de classe dans une sous-classe (*hiding*): à éviter

Les paramètres sont passés par valeur: c'est à dire qu'ils ont évalués, puis copiés dans la pile et c'est cette copie qui est récupérée par le code de la méthode. Un paramètre peut être marqué `final`. Les règles de promotion de type sont appliquées.

Constructeurs

Ce ne sont pas des “membres” de la classe (on ne qualifie pas un constructeur par rapport à une instance ou par rapport à une classe: on l’appelle par `new`).

Critères à controler lors de la définition :

- Nom : obligatoirement le nom de la classe.
  Noter que les constructeurs n’étant pas hérités on est obligé de les définir pour chaque niveau de la hiérarchie de classes: on a bien une correspondance nomClasse-nomConstructeur.

- Paramètres : mêmes principes de définition des paramètres que pour les méthodes.
  Les constructeurs peuvent être surchargés (plusieurs contructeurs possibles avec des paramètres différents).
  Une classe dépourvue de définition est dotée automatiquement d’un constructeur par défaut (constructeur sans paramètres) à condition que sa super-classe lui rende accessible un constructeur sans paramètres (implicite ou non).
  Dès qu’un constructeur est défini la classe perd son consturcteur implicite (on peut alors définir explicitement un constructeur sans paramètres).

- Structure interne : le bloc de définition a une architecture particulière:

  - une première instruction doit concerner l’appel d’un autre constructeur : soit de la même classe (par `this`) soit de la super-classe (par `super`). Si le constructeur à appeler est le constructeur sans paramètres de la super-classe cette instruction peut être implicite.

  - après cette première instruction le système évalue (dans l’ordre défini par le code source) toutes les initialisations explicites de variables d’instance (ou de blocs d’instance) -sauf si la première instruction est un appel au constructeur this-

  - Les autres instructions d’initialisation viennent ensuite. Si une variable membre de l’instance est marquée final sans avoir été initialisée, son initialisation doit être réalisée au travers du code de tous les constructeurs.

(par ailleurs éviter d'appeler dans le code du constructeur des méthodes de l'instance courante qui ne soient pas marquées `final` ou `private`).

- Résultats: Un constructeur ne doit déclarer aucun résultat (même pas `void`). Son rôle est d'allouer et initialiser une instance.
  Par contre un constructeur peut propager des exceptions et donc refuser de construire / initialiser une instance.

- Modificateurs : `public, protected, private`
  (pas de static, final, native, strictfp, synchronized)

- Cas particuliers :

  - On ne peut pas créer une instance d'une classe marquée `abstract` mais celle-ci peut tout de même disposer de constructeurs qui seront utilisées par les sous-classes par l'invocation de super() (ou au travers de la définition / invocation d'une classe anonyme)

  - classes "internes" :

    - on ne peut pas définir de constructeur dans une classe anonyme mais on peut invoquer le constructeur de la super-classe de la classe anonyme en lui passant des paramètres.

    - les constructeurs de classes membres d'instance on besoin d'être invoqué à partir d'une référence du type de la classe englobante dont la classe est membre.

    - cas extrême : il est possible dans la définition du constructeur d'une classe interne d'instance d'appeler `this` ou `super` comme membre d'un paramètre du type de la classe englobante:

```
class Interne2 extends Interne1 {
                    ....
          public Interne2(Englobante inst, int x){
                inst.super(x) ; //dans le contexte
                     // de l'instance paramètre
```

Blocs

Les blocs permettent de regrouper un ensemble d'instructions. Ils déterminent également des zones hiérarchisées de portée des variables:

```
{// bloc englobant
        int nombre ;
        .....
        { // bloc interne
                int autreNombre ;
                ....
                nombre = 10 ; // correct
        } // fin bloc interne
        ...
        autreNombre = 7 ;    // ERREUR ! IMPOSSIBLE
        nombre = 20 ;       // possible
}
```

Certains dispositifs syntaxiques permettent de déclarer également des variables dans la portée du bloc associé : paramètre de méthode ou de constructeur, paramètre d'un bloc `catch`, déclaration de variable dans la partie "initialisation" d'une boucle `for`...

En dehors des blocs associés aux déclarations de classe, aux "corps" des méthodes ou des constructeurs, aux structures de controles, aux blocs `try/catch/finally`, et aux blocs `synchronized` (hors périmètre du cours) il existe des blocs "isolés" : blocs d'initialisations statiques, blocs d'initialisations d'instance et blocs isolés dans un autre bloc.

Blocs statiques

Un bloc de premier niveau marqué `static` est évalué au moment de l'initialisation de la classe.

Il peut y avoir plusieurs blocs “d’initialisation de classe”: ils sont évalués dans l’ordre d’apparition dans le code.

```
public class Maintenance{
static URL urlMaintenance ;

   static {
         String url = System.getProperty("produitX.url") ;
         URL secours = null ;
         try {
            secours = new URL ("http://java.sun.com") ;//testé
            urlMaintenance = new URL(url) ;
         } catch (Exception exc)  {
            urlMaintenance = secours;
         }
   }

....
```

Blocs d'instance

Un bloc de premier niveau non marqué `static` est évalué au moment de la création de l'instance (après initialisation de la super-classe et avant l'exécution du reste du code du constructeur).

Il peut y avoir plusieurs blocs "d'initialisation d'instance": ils sont évalués dans l'ordre d'apparition dans le code. Ces blocs, rares, peuvent être utilisés soit pour réaliser des initialisations nécessitant des blocs `try/catch`, soit pour remplacer des opérations d'initialisation d'instance dans une classe anonyme (qui ne dispose pas de système de définition de constructeur).

Blocs non associés à une structure

Dans un bloc il est possible d'ouvrir un nouveau bloc de contexte. ceci permet d'isoler des déclarations de variables, d'améliorer la lisibilité et la structure du code.

```
...
{ // PANEL HAUT (nouveau bloc)
          panelHaut = new Panel(new FlowLayout(
                                  FlowLayout.LEFT));
          Button button1 = new Button("1") ;
          panelHaut.add(button1 ;
}
add(panelHaut, BorderLayout.NORTH) ;
```

Un tel bloc de code peut aussi être étiqueté :

```
__PANEL_HAUT__: {
          ....
          if(...) break __PANEL_HAUT__ ;
           //
          ....
}
```

Ces dispositifs (peu usités) peuvent être utiles pour attirer l'attention du lecteur du code sur des structures implicites :

Types primitifs scalaires, types objets

Il y a en Java deux grandes catégories de types qui se distinguent nettement par leur utilisation et leur allocation en mémoire:

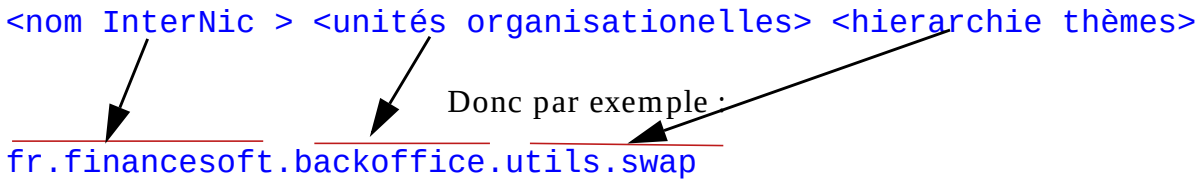
- Les types primitifs : ce sont des types prédéfinis qui représentent des valeurs scalaires (booléens, caractères, types numériques entiers et flottants). Leur représentation interne est standardisée et portable (exemple : `int` signés sur 32 bits en ordre poids fort-poids faible, `char` sur 16 bits non signés avec codes UTF16 Unicode).
  Ces types sont: `boolean, char, byte, short, int, float, double`.
  Dans le code d'une méthode ou d'un constructeur ces types sont alloués sur la pile d'exécution (avec une valeur indéterminée tant qu'ils ne sont pas initialisés). Une variable de type primitif désigne toujours une valeur.

- Les types objets : qu'ils soient définis par Java ou par l'utilisateur ils héritent de la classe `java.lang.Object`.
  Les instances sont créées par l'appel de `new Constructeur()` et sont systématiquement allouées dans le tas. Les champs primitifs sont initialisés à zero par défaut.
  Une variable de type objet désigne toujours une **référence** vers une instance en mémoire. Il n'est pas garanti que le type effectif de l'instance soit celle du type déclaré de la variable (polymorphisme).

  Un cas particulier d'objets sont les tableaux. Ce sont de vrais objets qui ne disposent que d'un seul membre immuable (`length`), mais qui permettent d'accéder à un ensemble de valeurs qui sont soit des scalaires (tableau de primitifs), soit des références (tableau d'objets -y compris tableau de tableaux-).

Packages, règles de responsabilité

Toute classe Java doit appartenir à un package. Un package est un regroupement de classes qui concourent à des objectifs communs ou analogues. Il est de bonne pratique d'organiser les packages selon une hiérarchie :



Les packages structurent les responsabilités des codes: chaque code a des possibilités d'interagir ou non avec d'autres codes en fonction des droits d'accès qui lui sont accordés:

- Un membre ou un constructeur marqué **`private`** est sous la responsabilité du code courant (on pourrait dire: sous la responsabilité du programmeur en charge du source courant). Il n'est accessible que dans le source courant.

- Un membre ou un constructeur sans modificateur (*package friendly*) n'est accessible que depuis les codes du même package (on pourrait dire: accessible par les programmeurs qui font partie de la même équipe de développement).

- Un membre ou un constructeur marqué **`protected`** est accessible depuis les codes du même package, de plus:

  - un membre `protected` est accessible comme membre hérité (c.a.d membre de l'instance courante) dans les classes qui héritent de la classe courante et qui sont situés dans d'autres packages.

  - un constructeur protected ne peut être invoqué qu'au travers de `super`(..) dans les classes qui héritent de la classe courante et qui sont situés dans d'autres packages.

- Un membre ou un constructeur marqué **`public`** est accessible dans tout code.

Les classes de "premier niveau" ne peuvent être que public ou de niveau package.

Aide-mémoire 20

Contenu:

- les outils du SDK
- aide-mémoire HTML pour javadoc
- glossaire
- adresses utiles

Le SDK

Divers sites permettent de télécharger un SDK (voir `http://java.sun.com/j2se/`)

Le SDK s'installe en deux phases :

- L'installation de la machine virtuelle spécifique à votre machine / système
- L'installation de la documentation (commune à toutes les plate-formes)

Une installation typique du répertoire "racine" de Java contient les répertoires ou fichiers:

- **`bin`** : exécutables de lancement des utilitaires liés à Java : `java`, `javac`, etc.
- **`jre`** : l'installation de la machine virtuelle proprement dite. Exécutables, librairies dynamiques, librairies "système" ou librairies d'extension de Java, ressources générales de configuration de l'exécution.
- **`lib`** : librairies utilitaires spécifiques à l'implantation (par ex. librairies Java des utilitaires comme le compilateur, javadoc, etc.)
- **`include`** : les ressources pour compiler du code natif lié à Java.
- **`demo`** : ressources pour des démonstrations de programmes et leur code source.
- **`docs`**: le répertoire de la documentation générale en HTML (fichier index.html)
- **`src.jar`** : les sources des librairies standard de Java dans une archive Jar. Pour consulter ces sources faire

```
jar xvf src.jar
```

qui crée un répertoire **`src`** contenant les fichiers source.

Les outils

La documentation des outils est liée à la livraison du SDK.

voir `docs/tooldocs/tools.html`

outre **java**, **javac**, **javadoc** quelques autres outils utiles pour débuter:

- **jar** : permet d'archiver un ensemble de binaires java (et d'autres fichiers associés). C'est une archive dans un format compressé (ZIP) qui peut être passée comme élément de CLASSPATH. C'est de cette manière que des librairies ou des applications Java sont livrées. Ce point est developpé dans notre cours "maitrise de la programmation java".

- **javap** : "décompile" un binaire Java en assembleur JVM. Pratique pour vérifier rapidement le véritable contenu d'un fichier ".class". Pour un vrai décompilateur (binaire->source) il existe des logiciels domaine public.

- **jdb** : debuggueur "rustique" en mode commande.

- **appletviewer**: permet de tester rapidement la présentation d'une applet.

- **native2ascii** : utilitaire de conversion de fichiers texte (selon le mode de codage des caractères)

- **rmic** : "Compilateur" de *stub* RMI(fonctionne par introspection sur les classes d'objets serveur).

- **rmiregistry** : *registry* RMI de reférence.

- **rmid** : "daemon" système permettant l'activation dynamique d'objets serveurs RMI (sert également de *registry*).

javadoc et HTML

Voir usage de javadoc dans la documentation spécifique : `docs/tooldocs/`*plate-forme*`/javadoc.html`.

Point particulier pour les auteurs francophones: le source Java de la documentation doit être lisible ce qui pose un problème pour les caractères accentués qui peuvent être différents selon les plate-formes de codage. On veillera à ce que le HTML généré soit portable (option -charset).

Pour HTML voir les descriptions de référence à `http://www.w3.org/MarkUp` et `http://www.w3.org/TR/html4`

Aide mémoire HTML

Structures principales

Eléments principaux de la structuration du fichier html : n'ont que peu ou pas d'intérêt pour la documentation javadoc puisque le programme génère automatiquement les grandes structures de présentation du document HTML.

Structure générale :

```
<HTML>
        <HEAD>
                titre interne
                <TITLE> titre externe </TITLE>
        </HEAD>

        <BODY>
                corps du document (texte à balises)
        </BODY>
</HMTL>
```

Hiérarchie des sections:

```
<H1> premier niveau de titre de paragraphe </H1>
          <H2> second niveau </H2>
                    <H3> niveau suivant </H3>
<H1> encore premier niveau </H1>
```

Paragraphes et listes

```
<!-- un commentaire -->
<P> un paragraphe
</P>
<P> un autre paragraphe
<BR> un retour à la ligne forcé
</P>
<BR>
<HR> <!------------ un filet horizontal------------------------>
```

Liste simple :

```
<UL>
        <LI> un élément de la liste </LI>
        <LI> un autre élément </LI>
</UL>
```

Liste numérotée:

```
<OL>
        <LI> premier item </LI>
        <LI> deuxième item, etc. </LI>
</OL>
```

Liste descriptive:

```
<DL COMPACT>
        <DT> entrée dans la liste </DT>
                <DD> description de l'entrée </DD>
        <DT> un autre terme </DT>
                <DD> sa description </DD>
</DL>
```

Citation de texte :

```
<BLOCKQUOTE>
        La culture c'est ce qui reste quand on a tout oublié
</BLOCKQUOTE>
```

Texte préformaté :

```
<PRE>
        for( int ix = 0; ix < tb.length; ix++) {
                System.out.println(tb[ix]);
        }
</PRE>
```

Tableaux

```
<TABLE >
        <TR> <!--ligne-->
                <TH> cellule de titre
                <TD> cellule normale
                <TD> cellule suivante
        <TR> <!-- ligne suivante>
                <TH>
                <TD>
</TABLE>
```

Hypertexte et éléments allogènes

Lien vers un autre document:

```
voir <A HREF="http://java.sun.com"> ICI </A>
```

Définition d'une destination de lien à l'intérieur d'un document:

```
<A NAME="cible"> c'est ici
```

utilisation dans une reférence hypertexte:

```
voir <A HREF="#cible"> dans le même document</A>
... <!-- dans un autre fichier HTML -->
voir aussi <A HREF="http://www.truc.fr/doc.html#cible"> là </A>
```

Images

```
voici une image : <IMG SRC="image.gif">
ou <OBJECT data="image.gif" type="image/gif">
```

Styles physiques

```
dans le texte lui même <B>ceci est en caractères gras</>
et <I>ceci en italiques</I>
<U>ceci est souligné</U>
et <TT> ceci en caractères à chasse fixe (comme fonte
"courier")</TT>
```

Glossaire

voir aussi : `http://java.sun.com/docs/glossary.html`
pour la version française des termes voir
`http://www.granddictionnaire.com`

Quelques définitions dans ce glossaire sont marquées “terminologie non-standard” car il n’y a pas un large accord sur la pratique terminologique en Français. Le terme décrit a toutefois été choisi pour être utilisé dans ce support de cours.

accesseur

(*accessor*) méthode permettant d’interroger une instance sur la valeur d’un attribut encapsulé. Par convention les accesseurs sont de la forme : `type getXXX()` . où XXX est le nom du champ. Voir “mutateur” et la convention correspondante.

adaptateur

(*adapter*) Dans le contexte particulier du traitement des événements en Java: une classe qui réalise toutes les méthodes d’une interface avec un code vide. On doit créer une classe qui hérite de l’adaptateur et qui redéfinit une ou plusieurs de ces méthodes.

agregation

En terme d’analyse : relation “qui possède” ou “ensemble/élément”. Relation de longue durée entre deux objets, mais relation non exclusive; le cycle de vie des deux objets n’est pas nécessairement lié.

allocation

Réservation d’un emplacement en mémoire. (par exemple pour contenir les données liées à un objet). En java on distingue les allocations “automatiques” (sur la pile: variables scalaires ou références), les allocations dans le tas (par `new`), le pool de constantes, et les allocations

des classes elle-mêmes par les ClassLoaders. Contrairement à d'autres langages on n'a, en Java, aucun accès à la gestion de ces emplacements mémoire.

`analyse`

Phase de définition de ce qui doit être fait en terme de "métier".

`API`

(*Application Programming Interface*) la liste des accès programmatiques à un objet d'une classe donnée. En pratique on a tendance à restreindre cette description aux membres et constructeurs `public` ou `protected` qui constituent le "contrat" de type défini par la classe vis à vis des utilisateurs externes à l'équipe de développement.

`applet`

(Littéralement "Appliquette" -petite application-). Code Java téléchargé et exécuté par un navigateur WEB.

`architecture à trois parties`

(*three tiers architecture*) Application(s) éclatée en trois parties distinctes (chaque partie étant éventuellement accessible au travers du réseau). Une partie est chargée de l'interaction utilisateur (avec parfois un peu de logique déportée), une partie centrale est chargée de régler la logique applicative et une autre est constituée du fond de données de l'entreprise.

`association`

En terme d'analyse : relation "qui utilise" . Relation entre deux objets liés par la réalisation d'un service.

`attribut`

Variable membre vue d'un point de vue conceptuel. On a ici une vue plus large puisqu'un attribut peut, par exemple, être en réalité dérivé par calcul (exemple "surface" sera calculé à partir d'autres informations gérées par l'objet)

automatique

Variables propres à un bloc de code -comme une méthode-(paramètres, variables locales). Ces variables (scalaires, références vers des objets) sont allouées sur la pile et ont une “durée de vie” qui ne dépasse pas le temps d’exécution du bloc de code (exception: les variables automatiques marquées “final” et copiées dans le cadre de classes locales).

bean

Composant réutilisable. Un bean est une classe Java sur laquelle on peut obtenir des informations par programme: on utilise pour celà les mécanismes standard de reflexion / introspection de Java, des classes de documentation (comme BeanInfo) et des conventions de nommage. Les informations ainsi obtenues permettent par exemple de personnaliser la mise en oeuvre du composant dans un contexte d’exécution , de générer du code de mise en oeuvre, etc. Il existe différentes conventions de Beans selon le type d’environnement de mise en oeuvre (Entreprise Java Beans pour les serveurs d’entreprise, Federated Beans pour l’administration de Systèmes, etc.)

bus logiciel

Système de propagation de messages par lequel un objet émetteur peut s’adresser à plusieurs objets recepteurs sans les connaître. Les objets recepteurs s’abonnent à un bus et l’émetteur s’adresse au bus (un peu comme une fréquence radio).

champ

voir “variable d’instance”

chargeur de classe

(*ClassLoader*) Une classe java qui permet de charger le code d’autres classes java . A chaque ClassLoader est associé un espace de noms et une politique de sécurité particulière aux URLs d’origine des codes qu’il gère. On peut donc retrouver la “même” classe dans des ClassLoaders différents ce qui peux donner des effets surprenants comme des variables partagées (statiques) ayant des valeurs différentes.

classe

Unité de regroupement de données et de codes (agissant éventuellement sur ces données). Certaines classes servent de matrices pour la fabrication d'instances sur ce modèle (de regroupement de données et de code) d'autres sont un simple regroupement de services (classe n'ayant que des méthodes de classe).

classe abstraite

Une classe qui est utilisée pour représenter un concept abstrait, et à partir de laquelle on ne peut pas créer d'objets. Les méthodes qui sont déclarées sans fournir de définition sont appelées méthodes abstraites .
Ces classes sont souvent le résultat d'une décision de mutualisation de services entre des classes analogues : les méthodes concrètes sont partagées et les services génériques sont décrits par des méthodes abstraites.

classe anonyme

Classe "sans nom" définie et utilisée à la volée dans le code. On crée de cette manière une instance ayant des propriétés *ad hoc* et pour laquelle on n'envisage pas de réutilisation.

classe d'encapsulation

(terminologie non-standard pour *wrapper class*) Classes liées à des types scalaires primitifs et permettant :

- de créer des objets immuables qui contiennent un scalaire du type correspondant (par exemple de type `int` pour la classe `Integer`);
- d'obtenir des services (méthodes de classe) liés à ces types scalaires (par exemple conversion). Si une méthode prend un argument de type générique `Object` et qu'on veut l'utiliser avec des types primitifs on met alors en place une convention qui permet de passer une instance du type d'encapsulation correspondant. Le champ `TYPE` de ces classes est un objet de type `Class` qui décrit le type scalaire associé.

classe interne

(terminologie non-standard: englobe ici les termes *inner class* et *embedded class*) classe définie dans la portée syntaxique du code d'une autre classe. On distingue les "classes membres" et les "classes locales".

classe locale

Classe définie dans la portée syntaxique d'un code exécutable. La classe peut utiliser des variables automatiques de ce code à condition qu'elles soient marquées `final`.

classe membre

Classe définie au niveau des déclarations à l'intérieur d'une classe. Elle devient un membre d'instance ou de classe avec les obligations et propriétés qui en découlent. Contrairement aux classes de "premier niveau" elles peuvent être `private` ou `protected`. Elles ont, a priori, accès aux champs et aux services qui sont dans la même "unité de compilation".

CLASSPATH

Liste des chemins d'accès aux classes. Peut contenir des répertoires, des archives Jar .

code natif

Une classe java peut faire référence à du code compilé pour un système et un processeur particulier. Il y a bien sûr des conventions particulières pour assurer la liaison avec ce code (voir J.N.I) et la classe perd sa portabilité inter-plateformes.

compilation conditionnelle

Dans certains langages le code source peut contenir des directives permettant de ne compiler certaines parties que sous condition (par exemple paramètre passé au compilateur). Ceci permet de fabriquer différentes version d'un logiciel: par exemple une avec des informations de trace et l'autre , plus performante, sans ces traces. Le code source de Java évite les systèmes de macro-commandes (pour des raisons de lisibilité) mais on peut réaliser des blocs de code qui ne sont pas compilés; la version 1.4 contient aussi des facilités d'assertions.

composition

En termes d'analyse : relation "est composé de".Relation de longue durée entre deux objets, relation exclusive; le cycle de vie des deux objets est lié au sens où si on supprime l'objet contenant on supprime l'objet contenu. Ex. La description du Client "contient" son Adresse.

conception

Phase permettant de définir, à partir de l'analyse, une architecture, des objets techniques qui conduiront au programme réel.

constante

Une valeur qu'on ne peut pas modifier. En Java la notion est relativement complexe, en effet on distingue :

- les constantes de *compile-time* (des primitifs scalaires essentiellement)
- les constantes évaluées au *run-time* (soit au moment de l'initialisation de la classe, soit au moment de la création de l'instance)

Il n'est pas garanti que les champs d'une constante objet soient eux-mêmes constants.

constructeur

Code d'initialisation qui accompagne l'allocation d'une instance.

container

Plusieurs sens possibles en Java :

- Composant graphique qui en "contient" d'autres. Un gestionnaire de disposition associé permet de disposer des composants à l'intérieur d'un container.
- Environnement de déploiement de Beans

conversion

(*cast*) Transformation effective d'une valeur scalaire d'un type donné dans un autre type . Conversions implicites (par exemple short->int), conversions explicites (par exemple int-> short); certaines conversions sont impossibles (par exemple int -> boolean). Dans ce document ce terme n'est pas utilisé pour les références d'objet car il n'y a pas de transformation de la valeur.

délégation

En termes d'analyse : agrégation permettant à un objet contenant de rendre un service en déléguant sa réalisation à l'objet contenu.

directive

Mot reservé servant à donner des indications de comportement au compilateur. Par exemple `package`, `import`. Les directives ne sont pas des instructions exécutables.

doclet

Code java dynamiquement associable à l'utilitaire `javadoc`. Ce code permettra de personnaliser les traitements associés à certaines structures de la documentation (par exemple mise en forme spéciale ou détection d'absence)

dynamicité

Capacité de Java de choisir du code exécutable au *run-time* : par polymorphisme , par chargement dynamique de classe.

encapsulation

Rendre inaccessible à du code des données ou des services situés dans un autre code.

espace de nom

(terminologie non-standard : décalque de *namespace*) En général un dictionnaire qui associe un symbole, un "nom" à une entité du langage. En Java il s'agit d'un dictionnaire de noms de classes et il en existe un par ClassLoader.

événement

Un objet qui rend compte du fait que "quelque chose s'est produit" au niveau d'un autre objet. Un autre aspect est lié aux conditions de propagation : idéalement l'événement n'est généré que s'il y a quelqu'un qui s'est inscrit pour le récupérer, implicitement le code de traitement ne doit pas bloquer le thread qui l'a déclenché et des mécanismes spéciaux peuvent permettre de fusionner des événements proches dans une série donnée pour éviter des engorgements. Pour d'autres types de propagation d'objets voir "exception" et "bus logiciel".

`exception`

Un objet qui rend compte d'une condition erronée. La propagation de ces objets a un mode particulier: l'émetteur ne connait pas le récepteur. Il doit y avoir en amont dans la pile courante un bloc de code chargé de filtrer les exceptions qui "remontent" la pile. On a ainsi une indépendance entre le code chargé de signaler l'exception et le code chargé de réagir. Un des aspects intéressant de Java est que les exceptions peuvent faire partie du "contrat" d'une méthode ou d'un constructeur et que ce contrat oblige les codes utilisateurs à récupérer les exceptions potentielles.

`expression`

Partie d'une instruction qui s'évalue à l'exécution (opération, appel de méthode, ...). L'évaluation d'une expression peut avoir des effets de bord (tel qu'une affectation à une variable), un résultat qui peut être utilisé à son tour comme opérande ou comme argument dans une autre expression, ou même avoir un effet sur l'enchainement des instructions. Comme en C/C++ l'affectation (opérateur =) est une expression qui rend un résultat (d'où les notations : `y =x =z;` ou `o.m(x=y)`)

`filtre`

En Java terme utilisé dans les mécanismes d'entrée/sortie. Un filtre est incorporé dans un enchainement de flots: il prend ses données dans un flot, transforme les données ou le fonctionnement du flot et rend ces données à un autre flot. Exemples : `BufferedInputStream` transforme "l'écoulement" du flot en bufferisant; `InputStreamReader` transforme un Inputstream en Reader (c'est un filtre bien qu'il ne dérive pas d'une classe filtre comme `FilterInputStream`)

Autre sens: mécanisme qui permet de sélectionner des objets parmi un ensemble d'objets en fonctions d'attributs, de propriétés, de nom, etc.. Exemple: `FileNameFilter`.

`flot`

Modèle abstrait d'entrée/sortie représentant les dispositifs de lecture/écriture comme une suite ordonée de données. On lit par exemple successivement des données et le flot continue de fournir ces données au point d'accès. Le modèle est puissant car il permet de représenter des lectures dans des situation très différentes : lecture dans un fichier, dans un tableau en mémoire, etc...

fonction

Java ne dispose pas de fonctions (ni de procédures) mais ses méthodes suivent un modèle fonctionnel c'est à dire elles définissent une expression de la forme: `res f(listeargs)`. Les arguments éventuels sont des paramètres en entrée et le résultat éventuel est constitué d'une seule valeur. Java adhère à un modèle avancé qui déclare également les résultats erronés potentiels (clauses `throws`).

forçage de type

(également utilisé: "projection de type"; terminologies non-standard: rendent un des sens de l'opérateur *cast*) Opération par laquelle on demande de considérer une référence d'un type donné comme une référence vers un type compatible. L'opération a des effets au *compile-time* (on peut déréférencer par rapport au type demandé) et au `run-time` (déclenchement d'une `ClassCastException` si le type effectif n'est pas compatible).

garbage collector

Tâche de fond (thread) de basse priorité qui recherche dans le tas des objets qui ne sont plus référencés (même indirectement) par les piles d'exécution. La mémoire est alors récupérée et, éventuellement "tassée".

gestionnaire d'événement

(*event handler*) Méthode spécialisée dans le traitement d'un objet événement et correspondant à une circonstance particulière (par exemple: `keyPressed` pour un `KeyEvent`)

gestionnaire de disposition

(*LayoutManager*) Classe chargée de la disposition graphique de composants à l'intérieur d'un Container.

gestionnaire de sécurité

(*SecurityManager*) Classe chargée de mettre en oeuvre les contrôles d'accès correspondant à une politique de sécurité. Il s'agit ici des controles liées à l'origine des classes et des droits des instances liées à ces classes.

glaneur de mémoire

(terminologie non-standard: extension du "glaneur de cellules" de Lisp. Autre terme: "ramasse-miettes") voir "garbage collector".

héritage

Deux aspects : conceptuel ou opérationnel.

Aspect conceptuel : relation “est un” ou “est une sorte de”. Peut découler d’une démarche de spécialisation (réutilisation par extension) ou de généralisation (on tente de capturer les points communs entre plusieurs classes)

Aspect opérationnel : les mécanismes d’héritage permettent de mettre en place des définitions différentielles (telle classe est définie par ses différences avec une autre classe, une partie est commune). Une nouvelle classe “hérite” des caractéristiques de sa classe “mère” (même si elle n’a pas toujours accès à tous les éléments -ceux qui sont privés par ex.-), elle peut redéfinir certains comportements ou en rajouter.

héritage multiple

Aspect conceptuel : une classe hérite de plusieurs autres classes distinctes.
Aspect opérationnel : n’existe pas en Java (permet ainsi de ne pas avoir de règles implicites complexes sur des comportements hérités: tout doit être codé explicitement).

Hotspot

(marque déposée) Compilateur auto-adaptatif couplant un compilateur JIT et un interprête sous le contrôle de règles d’optimisation. On obtient ainsi des performances remarquables à conditions d’être sous le contrôle de règles adaptées au type d’application (ex. les règles ne sont pas les mêmes pour un serveur ou pour un poste client graphique).

Ieee754

(IEEE = Institute of Electrical and Electronics Engineers) Standard de représentation et de fonctionnement pour les processeurs de calcul en virgule flottante.

implémenter

(terminologie non-standard: le mot n’existe pas en français mais il est retenu ici dans le sens très étroit de la déclaration du mot-clef *implements*) Dispositif par lequel une classe Java déclare connaître la sémantique d’une “interface” Java et s’engage à réaliser le contrat de type ainsi défini.

instance

Un objet: entité autonome créée sur le modèle de la classe auquel il appartient. L'objet est responsable de la réalisation de services qu'il rend en gérant son état interne de manière cohérente, éventuellement en collaboration avec d'autres objets.

interface

Le terme a de très nombreux sens en informatique (partout où se trouve une frontière conventionnelle sur laquelle se situent des échanges). En Java le terme à un sens particulier lorsqu'il désigne un type abstrait "pur" c'est à dire un type qui est défini par ce qu'il fait (une liste d'en-têtes de méthodes d'instance) et non par sa réalisation.

interface homme/machine

(I.H.M. ou en anglais *GUI -Graphic User Interface-* prononcer *gouwï)*Tout ce qui permet la communication utilisateur-ordinateur: plus particulièrement la présentation graphique et ses modes d'interaction.

introspection

voir "réflexion / introspection"

JIT

"Just In Time": exécuteur de pseudo-code qui compile à la volée ce code en code natif de la machine.

JVM

"Java Virtual Machine": norme décrivant le comportement d'un exécuteur du pseudo-code Java (et par extension tout exécuteur conforme à cette norme). Il s'agit en fait d'un "processeur" virtuel qui exécute un code binaire spécifique (le "pseudo-code"), la présence d'un tel émulateur sur une machine permet d'assurer la portabilité du code binaire Java.

liaison dynamique

(équivalent de "*virtual method invocation*" en argot C++) Mécanisme qui permet à l'exécution (au *runtime*) de choisir le "bon" code d'une méthode en fonction du type effectif de l'objet. En java le type effectif d'une référence peut être différent du type déclaré : soit l'objet est d'une sous-classe de la classe demandée soit il est d'une classe qui "implémente" le type d'interface déclaré. Le compilateur ne peut établir au *compile-time* quel est le code de méthode qui va être réellement exécuté

liaison statique

Le compilateur sait établir au *compile-time* quel est le code ou la donnée qui va être consulté (voir “liaison dynamique”). La redéfinition dans une sous-classe d’un membre qui est en liaison statique (“rupture du lien statique” ou *hiding*) rend parfois le code obscur.

linearisation

voir “serialization”

litteral

Notation qui permet de décrire une valeur. Pour les primitifs scalaires et les chaînes de caractères une notation qui décrit une constante : `12 12.3 'c' "chaine"` sont des littéraux. La déclaration
`Object[] tb = {new Object(), new Object()} ;`
utilise un tableau littéral.

mandataire

(*proxy*) code ou application qui reçoit des demandes adressées à un autre objet ou application pour les lui transmettre, les filtrer, etc.

membre

Ce qui est directement joignable en passant par une référence d’ instance ou une désignation de classe (cet “accès” se faisant par l’opérateur “.”). Les champs, les méthodes, les classes définies au niveau des déclarations sont des membres : soit des membres de classe (`static`) soit des membres d’instance (il en va de même pour les membres hérités des super-classes). Les constructeurs et les blocs de premier niveau ne sont pas des membres. Le terme est également valable pour les membres d’un package qui sont des classes ou des sous-package. Pour les tableaux sont membres : les membres hérités de `Object` et le champ `length`.

message

Le terme sert à exprimer le “découplage” entre une demande de service et sa réalisation: c’est à dire que celui qui demande un service ne sait pas a priori comment il va être rendu. Cela permet à la fois du contrôle de type (celui à qui on demande le service sait effectivement le rendre) et abstraction, dynamicité et maintenabilité (le code réalisant peut évoluer, le code réalisant peut ne pas être connu au *compile-time*).
Historiquement le terme vient du caractère interprétatif de certains langages à Objet: la requête était interprété et dans ces langages le polymorphisme recouvrait à la fois possibilité de redéfinition et de surcharge; en Java les deux concepts sont distincts et le polymorphisme n’implique que la redéfinition.méthode d’instance

Méthode appelée comme membre d’une instance (c.a.d `objet.methode()` ou `this.methode()`) et qui sert à modifier ou à tirer partie de l’état de l’instance et/ou à permettre d’avoir un comportement polymorphique.

méthode de classe

Méthode membre d’une classe (à appeler de préférence par `Classe.methode()`) qui est un service général lié à la classe mais qui n’a pas de raison d’être une méthode d’instance (pas de liaison logique avec une instance particulière, pas de recherche de polymorphisme).

modèle structurel

(terminologie non-standard pour rendre *pattern*) Dispositif abstrait décrivant un modèle d’assemblage de classes permettant d’obtenir une mécanique comportementale particulière. Exemple: le modèle `Observer/Observable` de Java décrit un classe “observée” pour laquelle des modifications seront signalées aux “observateurs”.

modèle/vue/controleur

Un “modèle structurel” qui propose de gérer en parallèle des données applicatives susceptibles d’être modifiées et leur(s) représentation (en général graphique). Imaginons, par exemple, un tableau de données qui sont à la fois modifiées par un processus externe (par ex. d’autres interfaces graphiques qui “partagent” les mêmes données à distance) et modifiées par une interface graphique locale: le modèle sera le tableau de données, la vue l’interface graphique et le controleur assurera les échanges et la cohérence entre les deux,

`mutateur`

(terminologie non-standard : décalque de *mutator*) Méthode permettant de modifier globalement sur une instance la valeur d'un champ dont l'accès est protégé. Par convention les mutateurs sont de la forme : `void setXXX(Type)` . où XXX est le nom du champ et"Type" le type du champ. Voir "accesseur" et la convention correspondanteNaN

(Not A Number) Dans les processeurs de calcul en virgule flottante des configurations de bits spéciales qui permettent de gérer des compte-rendus ou des résultats d'opérations particulières comme 0/0. Un NaN est absorbant pour toutes les opérations et toute comparaison (y compris avec lui même) rend faux. En Java voir les constantes `Double.NaN` et `Float.NaN` et méthodes `isNaN`.

`objet`

voir "instance"

`obsolescence`

(*deprecation*) Entre deux versions d'un programme Java une partie d'une A.P.I. peut être abandonnée (il apparaît, par ex., qu'à l'usage les spécifications sont fautives). Dans ce cas les méthodes (ou constructeurs) abandonnées peuvent faire toujours partie de l'A.P.I mais être marquée par la balise spéciale javadoc `@deprecated` . Le compilateur signale alors tout usage obsolete d'une A.P.I.

`package`

En Java regroupement de classes agissant dans un domaine particulier (les classes d'entrée/sortie, les classes graphiques, etc.). Les codes ainsi regroupés ont entre eux des responsabilités que n'ont pas les codes extérieurs au package.
En dehors de tout petits codes de démonstration il est exclu d'avoir des codes hors-package.
Pour éviter des conflits potentiels de nom il est d'usage de préfixer les packages avec le nom internet des organisations ("com.sun", "org.omg", etc.)

passage par référence

Dans certains langages de programmation on a la possibilité de passer un paramètre (par ex. à une fonction) qui reste dans l'espace du code appelant. Le code de la fonction peut alors éventuellement modifier la valeur et cette valeur est donc modifiée pour le code appelant. Ce type "d'effet de bord" n'est pas recommandé d'un point de vue théorique mais reste possible.
Java ne connait que le "passage par valeur" (ceci dit il est possible lorsqu'on passe une référence comme paramètre de pouvoir modifier un membre rattaché à cette référence et donc de produire un effet de bord).
Il arrive que l'on utilise le terme "passage par reférence" en Java lorsqu'on utilise `R.M.I` (invocation de méthodes distantes) et que l'on veut signifier que le paramètre reste un objet distant (`RemoteRef`).

passage par valeur

Le fait que les paramètres d'une méthode sont copiées sur la pile (à partir de valeurs connues dans le code appelant) et que toute modification de ces valeurs survenue dans le code de la méthode est sans effet sur le code appelant (en principe : voir "passage par réference").
Il arrive que l'on utilise le terme "passage par valeur" en Java lorsqu'on utilise R.M.I (invocation de méthodes distantes) et que l'on veut signifier que le paramètre est un objet qui va être "linéarisé" et va être copié dans la JVM qui exécute le code.

pattern

voir "modèle structurel"

persistence

Capacité de conserver l'état interne d'un objet y compris entre différentes instances de processus JVM. Un programme pourra par exemple "s'arréter" et retrouver l'état d'un objet persistant au redémarrage de l'application.

petit-boutien

(référence aux *voyages de Gulliver*: voir "poids fort / poids faible", utilisé également pour les formats de date jj / mm / aa)

pile

Partie(s) de la mémoire dans laquelle sont stockées les données nécessaires à l'exécution des méthodes (paramètres, variables locales). Ces données sont temporaires et sont gérées en "pile" c'est à dire qu'on les alloue au moment où l'on rentre dans l'exécution et qu'elles perdent toute signification au moment où l'on sort de l'exécution de la méthode courante.

poids fort/poids faible

Organisation de blocs de bits dans un processeur (par ex. pour la représentation d'un nombre entier). Dans de nombreuses architecture le "mot" mémoire est organisé de droite à gauche regulièrement (le bit le plus à droite a le "poids" le moins fort, et plus on va vers la gauche plus on représente des "poids" fort -hormis le bit de signe). Une telle architecture (processeurs Motorola, Sparc, entiers portables Java) est dite en ordre poids-fort / poids-faible (ou "gros-boutienne"). D'autres architectures (processeurs VAX, Intel) ont des blocs de bits qui sont regroupés de manière non homogène (de droite à gauche à l'intérieur du bloc mais de gauche à droite entre blocs) : elles sont poids-faible / poids-fort (ou "petit-boutienne").

pointeur

Dans certains langages type permettant de représenter une adresse en mémoire. Il n'y a pas de pointeurs en Java.

polymorphisme

Le terme a eu plusieurs acceptations selon le contexte informatique: généralement il s'agit d'un comportement générique qui s'adapte automatiquement au type effectif des variables.
En Java cette généricité concerne le point vue de l'exécution d'une méthode d'instance: pour une API donnée définissant un type: l'appel sur une référence d'une méthode (repérée par sa signature) peut correspondre à des codes différents en fonction du *type effectif* de l'objet (qui peut être différent du type déclaré soit par héritage, soit par "implémentation" d'interface).
Un tel découplage entre une demande de service et sa réalisation améliore la généricité, la maintenabilité et la dynamicité des programmes.

pool

Ensemble de ressources mises en commun. ex. pool de constantes mises en commun entre les classes.

portée

Capacité d'un code à accéder à une donnée référencée par un symbole. En Java la portée est liée à la portée syntaxique : dans un bloc on n'accède qu'aux variables du bloc local, à celles du bloc englobant (y compris de variables associées au bloc comme des paramètres). On accède aussi aux membres et constructeurs de la classe courante (ainsi qu'aux aux membres et constructeurs accessibles dans les autres classes) et à l'espace de nom des classes. Les modificateurs d'accès (public, private, etc..) sont, eux, liés à la notion "d'unité de compilation".

primitif

En java type scalaire (c.a.d non-objet) "cablé" dans le langage. Certains langages à objet n'utilisent pas de tels types et ne connaissent que des objets même pour représenter des données simples comme des entiers. De tels types existent en java (voir "classes d'encapsulation"), mais des soucis de performance (en particulier dans les calculs) ont fait préférer l'implantation de types primitifs.

propriété

Une association nom-valeur. Peut avoir plusieurs interprétations en Java:

- Une association de deux chaines dans un dictionnaire de type `java.util.Property` (l'une est la clef, l'autre la valeur). A de nombreuses utilisations comme recupérer des valeurs de configuration (`System.getProperties()`) ou des données paramétrées (`PropertyResourceBundle`).

- Dans un "bean": une association symbole-valeur découverte dynamiquement grâce aux conventions de notation et aux mécanismes de refléxion / introspection

pseudocode

(*bytecode*) Code "binaire" destiné à la machine virtuelle Java (voir "JVM"). Comme cette machine est, précisément, virtuelle il s'agit d'un pseudo code binaire.

récursivité

Possibilité pour une méthode de se rappeler elle-même dans sa définition.
Pour les amateurs :

```
public static long ackerman(long mx, long ny) {
        // parametres doivent etre positifs
        if( mx ==0L) {return ny+1; }
        if(ny==0L) { return ackerman(mx-1,1) ;}
        return ackerman(mx-1, ackerman(mx, ny-1)) ;
 }
```

redéfinition

(*overriding*) Reprise dans une sous-classe de la définition d'une méthode héritée de la super-classe. Cette redéfinition peux éventuellement faire appel à l'exécution de la méthode redéfinie (appel de `super.methode()`).
Avec la définition de méthodes abstraites (héritage de classes abstraites, réalisation de contrat d'interface) cette technique accompagne la liaison dynamique et le polymorphisme.
Ne pas confondre avec "surcharge" (*overloading*) et avec "rupture du lien statique" (*hiding*).

référence

En Java on n'accède à un objet qu'indirectement au travers d'une reférence: c'est à dire "quelque chose" qui permet de retrouver l'objet dans le tas. Contrairement aux apparences il y a peu de chances qu'une référence soit implantée comme un "pointeur" en mémoire (tel que ce type est utilisé dans certains langages comme Pascal ou C).

référence distante

Technique permettant d'avoir une référence sur un objet situé dans une autre J.V.M (voir R.M.I.)

référence faible

Technique permettant de gérer des caches mémoire. Certains objets que l'on sait reconstituer sont référencés "faiblement" c'est à dire que l'on autorise explicitement le glaneur de mémoire à les récupérer si on a besoin de mémoire, l'objet sera reconstitué si on en a besoin ultérieurement. (voir package `java.lang.ref`)

`reflexion/introspection`

Ensemble de mécanismes qui permettent d'inspecter une instance d'un type a priori inconnu et d'en connaître la classe, les membres et les constructeurs. On peut , par exemple, appeler dynamiquement une méthode inconnue au *compile-time* .

`rupture du lien statique`

(terminologie non-standard pour rendre le terme *hiding* dans son acceptation par JLS) redéfinition dans une sous classe d'un membre qui n'est pas soumis à évaluation dynamique. Exemple: redéfinition d'un champ ou d'une méthode statique.

`sandbox security model`

(pas de traduction connue: la plus appropriée nous semble être celle de l'*arène* -que ce soit pour des gladiateurs, des politiciens ou des toros on peut "jouer" dedans mais les occasions de sortie doivent être limitées- correspond d'ailleurs au sens étymologique de l'arène = sable). Modèle de sécurité du code appliqué par un gestionnaire de sécurité en l'absence d'une politique de sécurité configurée. En gros un code soumis à politique de sécurité ne peut consulter que quelques informations sur l'environnement (voir fichier `jre/lib/security/java.policy`), ne peut pas connaître ni accéder au système de fichier, ne peut pas déclencher un processus système local et ne peut pas faire un accès réseau (sauf avec le site dont le code est originaire).

`scalaire`

(terminologie non-standard empruntée à d'autres langages comme APL). Un type "non-objet" représentant une valeur atomique.
(En toute rigueur les types `long` et `double` ne sont pas strictement atomiques au regard de certaines opérations, voir chapitre de JLS "NonAtomic treatment of double and long").

`serialization`

"mise à plat" d'un objet pour envoi dans un flot spécial (ObjectStream). Utilisé pour la persistence ou pour la communication d'objets au travers du réseau.

spécification

Définition du comportement d'une entité informatique (objet, opérateur, structure de contrôle, ...). Pour prendre l'exemple des "interfaces" Java les spécifications sont rendues à la fois par des moyens formels analysables par informatique (les en-têtes de méthodes abstraites) et par la documentation javadoc qui décrit de manière informelle le "contrat" de comportement de ces méthodes.

statique

(abus de langage venant de la reprise -malheureuse- du mot clef `static` de C/C++) Qui appartient à une classe et non à une instance. Les membres de classes sont préfixés du modificateur `static` (il en est de même des blocs de classe).
Aspect avancé: l'allocation mémoire correspondante n'est en rien l'allocation statique présente dans d'autres langages: l'allocation est liée au chargeur de classe et donc il peut y avoir plusieurs fois la "même" classe en mémoire (et donc par exemple plusieurs versions d'une variable partagée "statique").

structure de contrôle

Elément du langage permettant de modifier le déroulement successif des instructions et de les regrouper par blocs: aiguillages (`if, switch`), boucles (`while, do-while, for`), captures (`try-catch-finally`), sections sous contrôle de concurrence (`synchronized`).

surcharge

(*overloading*) capacité de définir plusieurs méthodes avec la même intention sémantique et des paramètres différents (ex. `println` de `PrintStream` qui peut gérer des entiers, des flottants, des objets, etc.). Existe aussi pour des constructeurs.
Ne pas confondre avec "redéfinition" (voir aussi "message").

tas

Partie de la mémoire d'un processus qui sert à l'allocation dynamique de données dont le nombre n'est pas a priori connu.
En Java tout est pratiquement alloué dans le tas et le "glaneur de mémoire" gère la désallocation des objets qui ne sont plus referencés.

`thread`

(pas d'accord général sur la traduction du terme en français) Tâche élémentaire à l'intérieur d'une JVM. Ces processus légers peuvent être gérés par le système d'exploitation ou par l'exécuteur Java lui-même. Un ordonnanceur de tâches passe la main à une tâche particulière en fonction de priorités et aussi de stratégies qui ne sont pas spécifiées par le standard. La gestion cohérente de données qui sont accédées par plusieurs threads concurrents est un des aspects les plus difficile de la programmation en Java.

`transtypage`

(terminologie non-standard: rend les différents sens de *cast*) Terme générique regroupant "conversion" et "forçage de type".

`UML`

Formalisme de représentation de modèles (en principe indépendant des langages et des méthodologies). Utilisé ici essentiellement pour montrer des diagrammes de classes.

`variable d'instance`

Variable "membre" d'une instance. Déclarée au premier niveau de la classe elle est accessible essentiellement par les méthodes de l'instance courante (mais pas par les méthodes de classe), La combinaison des valeurs des variables d'instance constitue "l'état" de l'instance.

`variable de classe`

voir "variable partagée".

`variable partagée`

Variable "membre" d'une classe. Déclarée au premier niveau de la classe (avec modificateur `static`) elle est accessible au travers de la désignation de la classe (et,mais c'est moins courant, au travers de référence à une instance de la classe).

`veilleur`

(terminologie non-standard: idée de *Listener*) Objet qui s'abonne pour recevoir des événements et dont la classe implante des gestionnaires d'événements.

XML

Formalisme permettant de représenter des données sous forme textuelle. Il s'agit d'un langage "à balises" (les balises étant du texte spécial qui documente et structure le texte lui-même). Ces données sont ainsi portables et aussi bien lisibles par une application que par un être humain.

Adresses utiles

Java

site fondamental: `http://java.sun.com` ;
+ http://www.ibm.com/java

spécifications : `http://java.sun.com/docs/books/jls/`

spécifications JVM : `http://java.sun.com/docs/books/vmspec/`

Un livre: "*Le langage Java*" (Arnold, Gosling, etc. ed. Vuibert ISBN 2-7117-8671-4)

Approche "objet"

Un livre: "*Object-Oriented Software Construction, Second Edition* " (Bertrand Meyer - ed. Prentice-Hall-)

UML

`http://uml.free.fr` (en français!)

Unicode

`http://www.unicode.org`

HTML et XML

`http://www.w3.org`

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z

Sun Microsystems France

La couverture des agences de Sun France permet de répondre à l'ensemble des besoins de nos clients sur le territoire.

Table 20.1 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 de Issy les Moulineaux
Le Lombard
143, avenue de Verdun
92442 ISSY-LES-MOULINEAUX Cedex
Tél : 01.41.33.17.00
Fax : 01.41.33.17.20

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 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

Bureau de Grenoble
32, chemin du Vieux Chêne
38240 MEYLAN
Tél : 04.76.41.42.43
Fax : 04.76.41.42.41

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 Lyon
Immeuble Lips
151, boulevard de Stalingrad
69100 VILLEURBANNE
Tél : 04.72.43.53.53
Fax : 04.72.43.53.40

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 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

Rapport d'erreur

Malgré tous nos soins des erreurs (ou des fautes d'orthographe!) sont peut-être encore présentes dans ce support. Merci d'utiliser cette page pour nous les signaler en mettant le numéro de chapitre et le numéro de page (remettre ensuite le document à l'animateur, merci!).