Programmation par objets, parallèle et répartie en Java

Juin 2005

P. Le Certen,
L. Ungaro

Copyright © P. Le Certen, L. Ungaro - Istic - Université de Rennes 1
La reproduction non autorisée d'une œuvre protégée constitue un délit.

PROGRAMMATION PAR OBJETS, PARALLELE ET REPARTIE EN

Java

Pascale LE CERTEN
lecerten@irisa.fr

Lucien UNGARO
ungaro@irisa.fr

juin 2005

IFSIC - Campus de Beaulieu - 35042 Rennes Cedex
Tel. 02 99 84 71 00 - fax 02 99 84 71 71
http://www.ifsic.univ-rennes1.fr

Table des matières

Ouvrage de référence :

Thinking in Java - Bruce Eckel
http://www.EckelObjects.com/javabook.html

Programmation par objets

en Java

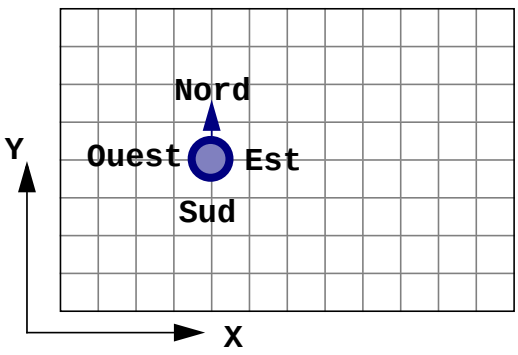
1 Classes et objets

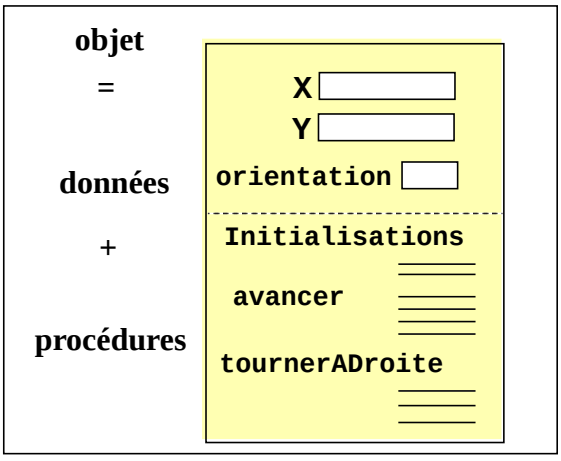
1.1 Introduction

La programmation par objets consiste essentiellement à structurer les applications selon les types de données. L'expérience montre que l'on obtient ainsi des logiciels fiables, évolutifs et faciles à maintenir. Le programmeur par objets se pose en premier la question : quels sont les objets fondamentaux de l'application ? Quelles sont les choses et les concepts qui constituent le domaine de l'application ? Il décrit alors chaque type d'objet sous forme de variables qui caractérisent son état puis il s'intéresse aux opérations que peuvent subir ces objets et les décrit sous forme de procédures.

Ceci conduit à une modularité basée sur les types de données. Les traitements sont toujours définis à propos de la définition d'un type de données, et non pas isolément comme c'est le cas pour une modularité basée en priorité sur la décomposition des traitements. Par exemple, pour une gestion de réseau ferroviaire, le programmeur par objets sera tenté de définir les objets suivants : trains, voitures, lignes, gares, ... Pour chaque type d'objet, il définira les propriétés de ces objets et les actions que l'on peut leur faire subir : un train est formé d'une collection de voitures, on peut accrocher une voiture, décrocher une voiture, chaque voiture a une capacité ...

Considérons un exemple très élémentaire : un robot (simpliste). Un tel robot occupe une certaine position (X,Y), il a une orientation parmi {Nord, Est, Sud, Ouest}, il est initialisé avec une position et une orientation données, il peut tourner à droite, il peut avancer d'un pas ... On le décrira par une association de variables d'état et de procédures. C'est une telle association de données et de procédures que l'on nomme habituellement un objet.





© IFSIC - Université de Rennes 1 - 2005

Il existe de nombreux langages permettant la programmation par objets : **Simula**, **Smalltalk**, **Eiffel**, **Turbo Pascal Objet**, **C++, Java**. Ces langages ont en commun les notions suivantes :

- La notion d'***objet***, qui possède deux aspects totalement indépendants :
  - Un objet est une ***chose dotée d'une identité*** (on peut le référencer). Ceci s'oppose à la notion de ***valeur*** (par exemple 12 ***n'est pas un objet***).
  - Un objet peut, accessoirement, être composé de données et de procédures liées à ces données : les données de l'objet constituent le contexte de travail pour les procédures associées.

- La notion de ***classe***, qui n'est autre que celle de type (ou modèle) d'objets,

- La notion d'***héritage*** : l'héritage permet de définir des types d'objets plus précis à partir de types d'objets plus généraux.

Si la programmation par objets se limitait à la notion d'objet et de classe, elle ne serait qu'un simple conseil méthodologique et ne nécessiterait pas vraiment de langage particulier. Il est toujours possible en effet, dans un langage tel que Pascal ou C, de regrouper les textes de définition des types de données (structures) et des procédures qui accèdent aux données de ces types.

En revanche, la notion d'héritage est plus profonde et ne peut être réalisée simplement par discipline, elle doit être offerte par le langage. Elle permet de définir une hiérarchie de types, des plus généraux aux plus spécialisés. L'héritage et les notions qui lui sont associées, notamment les procédures virtuelles, permettent la conception de logiciels extensibles et favorisent la réutilisation de modules logiciels existants.

Dans les paragraphes qui suivent, nous présentons les notions de programmation par objets du langage Java.

1.2 Définition d'une classe

La notion de ***classe*** est un enrichissement de la notion usuelle de ***type***. Une classe permet de définir un type d'objet, éventuellement compliqué, associant des données et des procédures qui accèdent à ces données. Les données se présentent sous forme de champs désignés par identificateurs et dotés d'un type. Ces champs sont généralement des variables qui représentent l'état de l'objet.

Les procédures, également appelées ***méthodes***, définissent les opérations possibles sur un tel objet. Ces données et procédures sont qualifiées de ***membres*** ou ***composants*** de la classe.

Nous prenons comme exemple de déclaration de classe, les "robots" précédents.

© IFSIC - Université de Rennes 1 - 2003

```
class Robot {

public static final int Nord = 1;    (constantes)
public static final int Est = 2;
public static final int Sud = 3;
public static final int Ouest = 4;

public int X;                        (variables d'état)
public int Y;
public int orientation;

public Robot(int x, int y, int o) {  (initialisation)
X=x; Y=y; orientation=o;
}
public Robot() {                     (autre initialisation possible)
    X=0; Y=0; orientation=Nord;
}
public void avancer() {
        switch (orientation) {
        case Nord : Y=Y+1; break;
        case Est  : X=X+1; break;
        case Sud  : Y=Y-1; break;
        case Ouest: X=X-1; break;
        }
}

public void tournerADroite(){
        switch (orientation) {
        case Nord : orientation=Est  ; break;
        case Est  : orientation=Sud  ; break;
        case Sud  : orientation=Ouest; break;
        case Ouest: orientation=Nord ; break;
        }
}}
```

L'attribut **`public`**, appliqué à un composant quelconque, donnée ou méthode, signifie que ce composant est désignable par son identificateur depuis le texte de toute autre classe. L'attribut **`final`**, lorsqu'il est appliqué à un composant qui est une donnée, signifie que ce composant est constant.

Initialisation des objets : *constructeur*

Une classe peut définir une ou plusieurs procédures d'initialisation appelées ***constructeurs***. Ces procédures ont le même nom que la classe, et s'il y en a plusieurs, elles se distinguent par le type ou le nombre des paramètres :

```
Robot(int x, int y, int o)
Robot()
```

Si l'on ne définit pas explicitement de constructeur, Java offre un constructeur par défaut, qui est sans paramètre, et qui ne fait pas grand chose (pour une classe héritière il se borne à appeler le constructeur sans paramètre de la classe de base : voir plus loin le chapitre sur l'héritage).

© IFSIC - Université de Rennes 1 - 2005

1.3 Déclaration et création d'objets

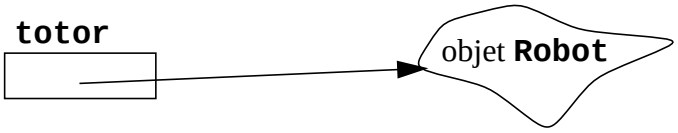
Une définition de classe constitue un modèle d'objet dont on peut ensuite créer autant d'exemplaires que l'on veut (on dit également ***instances***). En Java, la ***déclaration*** et la ***création*** des objets sont deux choses séparées. La déclaration consiste à définir un identificateur et lui associer un type afin que cet identificateur puisse désigner un objet de ce type. Par exemple, la déclaration des identificateurs **`totor`** et **`vigor`** pour désigner des objets de type **`Robot`** s'écrit :

```
Robot totor; Robot vigor;
```

En Java, comme dans de nombreux langages à objets tels que Eiffel, Pascal-Delphi ou Smalltalk, un identificateur de variable de type classe est associé à une ***référence*** qui désigne un objet. Une référence est l'adresse d'un objet, mais contrairement à un pointeur (tel qu'on en trouve en C, C++ ou Pascal) la valeur d'une référence n'est ni accessible, ni manipulable : elle ne peut que permettre l'accès à l'objet qu'elle désigne.
La déclaration seule ne crée pas d'objet. Elle associe l'identificateur à une référence appelée **`null`** qui ne désigne rien.

Pour créer un objet de classe ***C***, il faut exécuter **`new`** en citant un constructeur de la classe ***C***, avec ses paramètres. La primitive **`new`** rend en résultat la référence sur l'objet créé, qu'il suffit alors d'affecter à la variable déclarée :

```
...; totor = new Robot(0,0,Nord);
...; vigor = new Robot(5,12,Sud);
```



Il est possible de mêler déclaration et création :

```
Robot totor = new Robot(0,0,Nord);
Robot vigor = new Robot(5,12,Sud);
```

1.4 Utilisation des objets

Les composants d'un objet (champs de données et méthodes) sont désignés au moyen d'une notation pointée :

`totor.X;`	abscisse de **`totor`**
`vigor.avancer();`	fait avancer **`vigor`**
`totor.tournerADroite();`	fait tourner **`totor`**

© IFSIC - Université de Rennes 1 - 2003

1.5 Rédaction des méthodes

1.5.1 Déclaration de méthode

En Java, les textes des méthodes sont rédigés à l'intérieur de la définition d'une classe. Une déclaration de méthode a la forme suivante :

`T PPP (T1 p1, T2 p2, ...) {...` ***Corps*** **`...}`**

`T` est le type du résultat (**`void`** si la méthode ne rend pas de résultat).
`PPP` est le nom de la méthode.
`T1 p1, T2 p2, ...` est la liste des paramètres avec leur type. Si une méthode n'a pas de paramètre, il faut tout de même conserver les parenthèses : **`T PPP()`**.
Corps est composé de déclarations de variables et d'instructions. Les déclarations peuvent être mêlées aux instructions.

Le rendu de résultat se note **`return`** ***expression*** :

```
boolean superieur (float x, float y) {return x>y;}
```

On peut donner le même nom à plusieurs méthodes, à condition qu'elles diffèrent par le nombre ou le type des paramètres.

1.5.2 Instance courante : `this`

Dans le cas où un appel de méthode porte sur un objet, cet objet est appelé communément l'***instance courante***. Au sein des méthodes, les composants de l'instance courante sont directement désignables par leur identificateur.
Par exemple, lors de l'exécution de **`vigor.avancer()`**, **`X`** et **`Y`** désignent les composants **`X`** et **`Y`** de **`vigor`**.

Citation explicite de l'instance courante : `this`

Dans certains cas, on peut avoir besoin de citer explicitement l'instance courante dans le texte d'une méthode, par exemple pour passer cette instance en paramètre de procédure. On dispose pour cela du mot clé **`this`** (littéralement "lui").
Prenons un exemple classique, le placement de boutons dans une fenêtre de dialogue. La procédure de placement peut être définie dans une classe à part, **`Placement`**.

```
class Placement {
...
public static void placer(Fenetre f,Bouton b,int X,int Y)
{...}
}
```

La classe **`Fenetre`** ci-dessous place un bouton au sein d'elle-même en se passant en paramètre à la méthode **`placer()`** :

```
class Fenetre {
Bouton b=new Bouton();
...
Fenetre() { ... Placement.placer(this, b, 10, 30) ... }
...
}
```


© IFSIC - Université de Rennes 1 - 2005

1.6 Composants non liés à des objets : `static`

On a parfois besoin de procédures non associées à un objet. Une telle procédure, qualifiée de statique, doit être rédigée dans une classe, car il n'y a pas de procédures isolées en Java. Cela est souvent naturel, car la classe joue alors un rôle structurant en regroupant les procédures autour d'un même thème. Par exemple, on peut vouloir rédiger une procédure qui compare deux objets de type **`Robot`** et rend le meilleur des deux, mais sans vouloir associer la méthode **`leMeilleur`** à l'un plutôt qu'à l'autre. Pour cela, il faut déclarer la procédure avec l'attribut **`static`**.

```
class Robot {
public int solidite;
...
public static Robot leMeilleur(Robot r1, Robot r2) {
  if (r1.solidite > r2.solidite){return r1;}
  else {return r2;};
}}
```

Pour utiliser une telle procédure, il faut préfixer son nom par celui de la classe :
`Robot.leMeilleur(totor, vigor);`
Une méthode statique n'ayant pas d'objet courant, **`this`** n'y est pas défini.

On peut définir également des données qui ne sont pas des composants d'objets. Ces données sont déclarées avec l'attribut **`static`**, et chacune n'existe qu'en un seul exemplaire pour toute la classe.

Comment définir une simple procédure en Java

Il n'y a pas de procédures isolées en Java, il n'y a que des méthodes de classe. Pour avoir une procédure, on est donc obligé d'inventer artificiellement une classe. On peut adopter la convention suivante : donner un nom significatif à la classe, et un nom muet à la méthode, par exemple **`_()`** ou **`p()`**. Ainsi pour rédiger une procédure qui teste si un entier est pair :

```
class EstPair{
public static boolean p(int n) {return (n%2)==0;}
}
```

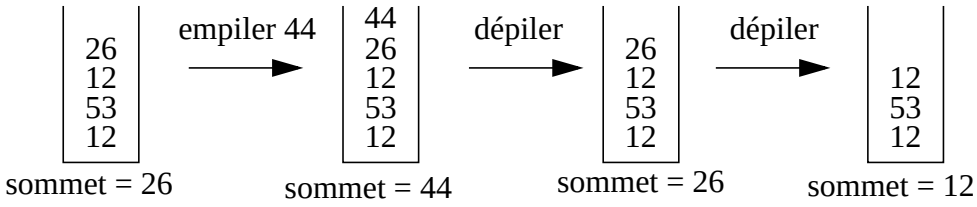
appel : **`EstPair.p(12);`**

1.7 Abstraction de la représentation : `private` et `public`

On utilise généralement une classe pour réaliser un type de données ou bien un comportement d'objet dont on souhaite cacher la mise en œuvre à l'utilisateur, afin d'assurer une meilleure fiabilité au logiciel : en effet, ceci interdit l'utilisation abusive de propriétés qui n'appartiennent pas vraiment au type de données, localise l'effet des erreurs et autorise le concepteur de la classe à modifier sa mise en œuvre, pour des raisons d'efficacité par exemple, sans rien changer au reste de l'application.
Prenons l'exemple simple du type de données "pile d'entiers". L'état d'un objet de ce type est une suite finie, éventuellement vide, de valeurs entières qui ont été empilées une à une. Le sommet est la plus récente des valeurs empilées, non encore retirées. On peut empiler une valeur. On peut également dépiler, ce qui retire le sommet courant.

© IFSIC - Université de Rennes 1 - 2003

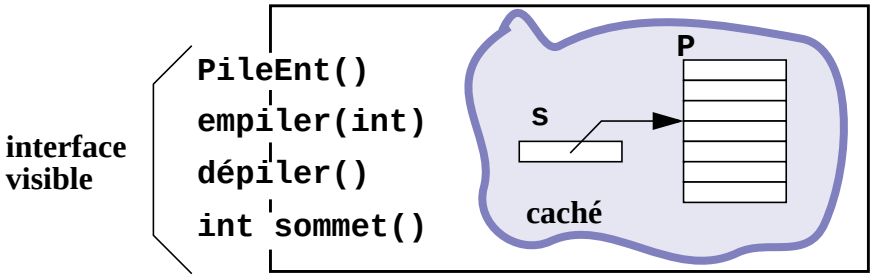
.



Ce type peut être défini au moyen d'une classe **`PileEnt`** :

```
class PileEnt {
public int s;
public int[] P = new int[100]; // tableau de 100 éléments
public PileEnt() {s=-1;}
public void empiler(int e) {s=s+1; P[s]=e;}
public void depiler() {s=s-1;}
public int sommet() {return P[s];}
}
```

La classe ci-dessus laisse voir trop de choses à l'utilisateur. Le tableau **`P`** et l'index **`s`** ne font pas partie des propriétés d'une pile, ce ne sont que des éléments d'une mécanique interne particulière. On pourrait réaliser la pile autrement, par exemple au moyen d'une liste de maillons chaînés. Les programmes qui utilisent la classe **`PileEnt`** doivent être indépendants de ces choix arbitraires de mise en œuvre.



Dans la définition d'une classe, pour rendre inaccessible un membre par son identificateur, on le déclare à la suite du mot clé **`private`**.

```
class PileEnt {
private int s;
private int[] P = new int[100];
public PileEnt() {s=-1;}
public void empiler(int e) {s=s+1; P[s]=e;}
public void depiler() {s=s-1;}
public int sommet() {return P[s];}
}
```

Avec cette définition, les utilisateurs de la classe **`PileEnt`**, c'est-à-dire les morceaux de programme qui déclarent, créent ou utilisent des objets de cette classe, n'ont directement accès qu'aux méthodes **`empiler`**, **`dépiler`**, **`sommet`** et au constructeur **`PileEnt`**. Seuls les textes qui définissent la classe ont le droit d'utiliser **`s`** et **`P`**.

© IFSIC - Université de Rennes 1 - 2005

En Java les classes peuvent être regroupées en ***paquetages*** (voir plus loin). Si on ne précise aucun paquetage particulier, les classes sont placées dans le ***paquetage par défaut***. S'il n'est pas qualifié **`private`**, un composant est accessible depuis tout le paquetage auquel appartient sa classe (il est donc par défaut ***public*** dans son paquetage), mais inaccessible depuis les autres paquetages.
Pour rendre un composant accessible depuis tous les paquetages, il faut le déclarer à la suite du mot clé **`public`** .

1.8 Classes internes

Java permet de définir une classe à l'intérieur d'une autre classe. Une telle classe est dite ***interne*** (***inner class*** en anglais). L'intérêt d'avoir une telle classe interne réside dans la modularité que cela apporte : le nom de la classe interne est purement local et cette classe peut même être rendue invisible depuis l'extérieur de la classe englobante si on la déclare avec l'attribut **`private`**.

Par exemple, pour programmer une pile au moyen de maillons chaînés, on peut définir une classe **`Maillon`** interne à la classe **`Pile`**.

```
class PileEnt {

private static class Maillon {
  int elt; Maillon suivant;
  Maillon(int e, Maillon s) {elt=e; suivant=s;}
}

private Maillon sommet;
public PileEnt() {sommet=null;}
public void empiler(int e) {
  sommet=new Maillon(e,sommet);
}
public void depiler() {sommet=sommet.suivant;}
public int sommet() {return sommet.elt;}
}
```

À l'extérieur de la classe englobante, on peut citer une classe interne (non privée) sous la forme : ***classe englobante*.*classe interne***

Dans le cas simple ci-dessus, la classe interne ne fait référence à aucun membre de la classe englobante : une telle classe interne est dite statique, et on la déclare dans ce cas avec l'attribut **`static`**.

Dans le cas général, une classe interne utilise les composants de l'objet courant de la classe englobante : il y a alors pratiquement une nouvelle classe interne par objet de la classe englobante, chacune étant liée à un objet. La création d'un objet de la classe interne doit dans ce cas citer un objet de la classe englobante.
Voici un exemple, simple mais illustratif. Nous voulons doter la pile d'un moyen de parcours de ses éléments, de telle sorte que plusieurs parcours puissent se dérouler en même temps sur une même pile. Une façon élégante de résoudre ce problème consiste à définir une classe **`Parcours`** associée à la pile et de créer un objet de cette classe

© IFSIC - Université de Rennes 1 - 2003

chaque fois que l'on veut réaliser un parcours. Le constructeur **`Parcours()`** initialise le parcours à partir du sommet, **`suivant()`** passe au suivant dans la pile, **`element()`** rend l'élément courant du parcours et **`estEnFin()`** teste si le parcours est fini.

```
class PileEnt {
private int s; private int[] P = new int[100];
public PileEnt() {s=-1;}
public void empiler(int e) {s=s+1; P[s]=e;}
public void depiler() {s=s-1;}
public int sommet() {return P[s];}

public class Parcours {
  private int courant;
  public Parcours() {courant=s;}
  public int element() {return P[courant];}
  public void suivant(){courant--;}
  public boolean estEnFin(){return courant==-1;}
}
}
```

Exemple d'utilisation :

```
PileEnt p= new PileEnt(); ...
// deux parcours sur p :
PileEnt.Parcours parc1= p.new Parcours();
PileEnt.Parcours parc2= p.new Parcours();
parc1.element(); parc1.suivant();
parc2.element(); parc2.suivant();
```

Noter la forme (surprenante) de la création d'un objet instance d'une classe interne :

objet*`.new` *constructeur

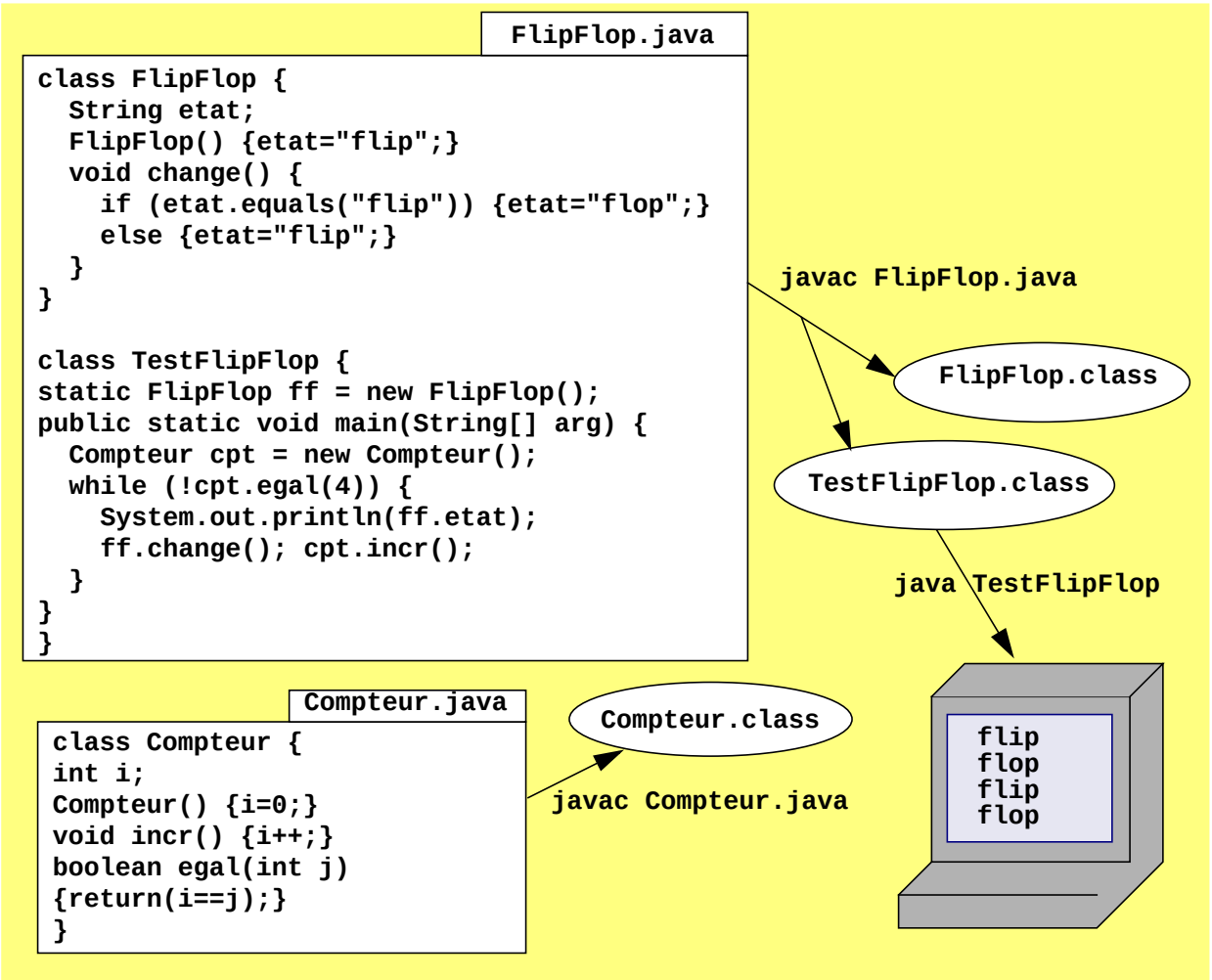
© IFSIC - Université de Rennes 1 - 2005

2 Structure et environnement des programmes

2.1 Structure des fichiers sources

Un programme Java consiste en une collection de définitions de classes. Un fichier source (suffixe **`.java`**) peut contenir plusieurs définitions de classes et les classes d'une même application peuvent être définies dans des fichiers sources différents.
La compilation produit un fichier objet par classe (suffixe **`.class`**). Le compilateur se lance par la commande : **`javac` *`fichier`*`.java`**

Les fichiers objets **`.class`** ne sont pas des programmes pour une machine particulière : ils sont codés en un langage destiné à être interprété. Ceci facilite la portabilité, notamment par transmission à travers les réseaux, de programmes Java dans des documents HTML.
Il n'y a pas d'édition de lien au sens habituel. Pendant l'exécution, les fichiers **`.class`** sont recherchés et chargés depuis le système de fichiers selon certaines règles de recherche. Une des classes (**`TestFlipFlop`** dans l'exemple ci-dessous) constitue le démarrage de l'application. Cette classe doit définir une méthode **`main()`** qui est appelée au lancement, par la commande : **`java TestFlipFlop`**



Le profil de **`main()`** est : **`public static void main(String[] arg)`**
le paramètre **`arg`** est un tableau de chaînes de caractères. Ce tableau contient les chaînes accompagnant (éventuellement) la commande de lancement.

© IFSIC - Université de Rennes 1 - 2003

Les commentaires

Les commentaires peuvent être notés de trois façons :

- soit entre **`/*`** et **`*/`**, qui est la forme héritée de C : **`/* blabla */`**
- soit entre **`/**`** et **`*/`**, commentaires de documentation, transformables par le logiciel Javadoc pour produire une documentation HTML : **`/** blabla */`**
- soit entre **`//`** et la fin de la ligne : **`// blabla`**

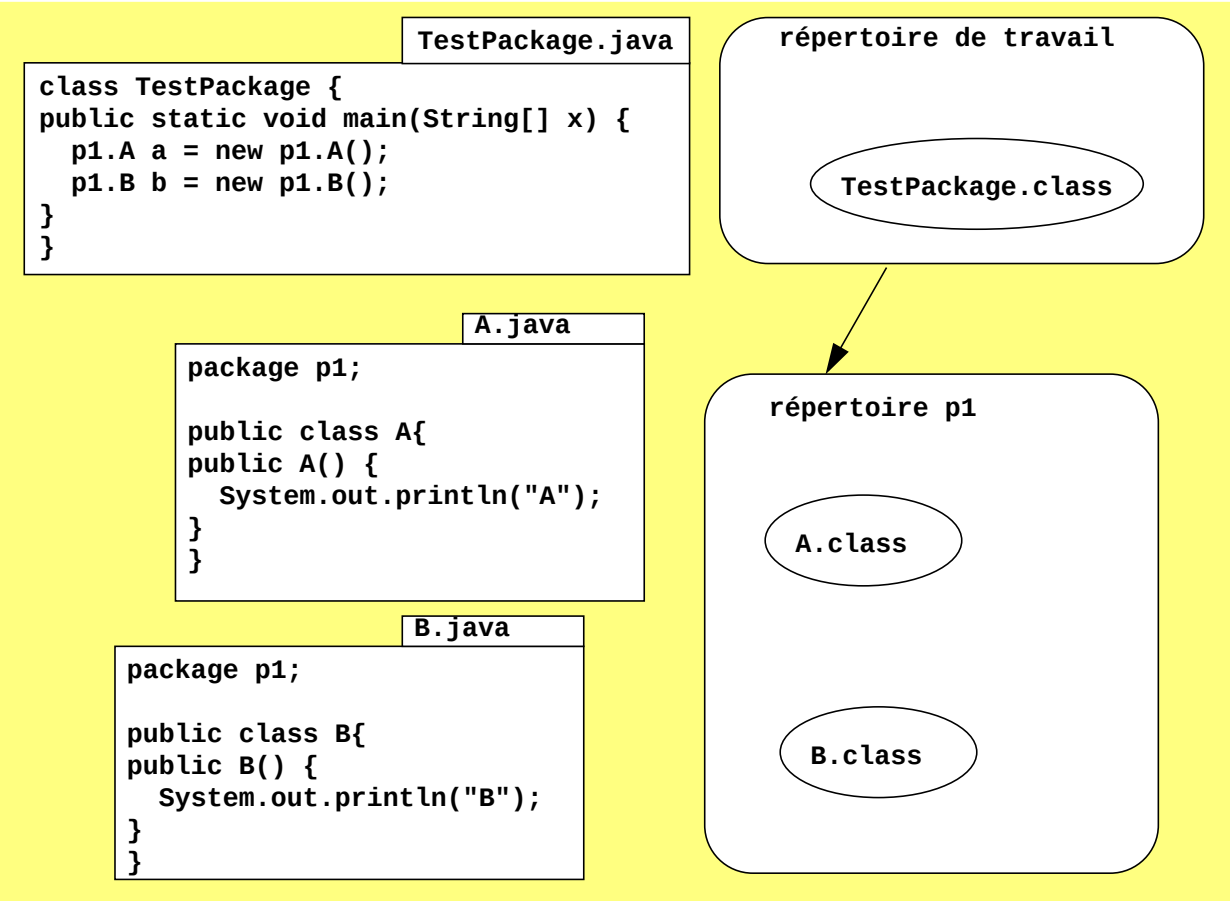
2.2 Paquetages : `package`

Les classes peuvent être regroupées en paquetages. Les classes d'un même paquetage sont dispersées dans plusieurs fichiers sources dont la première ligne est :

`package` ***nom-de-paquetage*****`;`**

Au sein d'un paquetage, on a accès aux classes de ce paquetage et aux classes déclarées **`public`** des autres paquetages, mais dans ce dernier cas il faut utiliser un nom absolu : ***nom-de-paquetage*****`.`*****nom-de-classe***
De plus les classes compilées d'un paquetage doivent être placées dans un répertoire de même nom que le paquetage.

Il est possible de désigner les classes par leur nom court, sans préciser le paquetage, à condition d'utiliser la directive **`import`** :
`import` ***nom-de-paquetage*****`.`*****nom-de-classe***
ou **`import`** ***nom-de-paquetage*****`.*`** pour désigner toutes les classes du paquetage.



Remarque : un fichier source doit comporter au plus une classe **`public`** et doit avoir le même nom que la classe **`public`** s'il y en a une.

© IFSIC - Université de Rennes 1 - 2005

Les répertoires où Java doit effectuer les recherches de paquetages sont généralement indiqués par une variable d'environnement. Sous UNIX cette variable d'environnement s'appelle **`CLASSPATH`**, et elle contient les noms de répertoires sous la forme suivante :

```
/usr/local/java/jdk1.1.4/lib/classes.zip:/home/junon/d03/
maitres/dupont/JAVA:.
```

Dans cet exemple, **`/usr/local/java/jdk1.1.4/lib`** est le répertoire qui contient les paquetages de la bibliothèque Java sous forme d'un fichier compressé **`classes.zip`**. Le répertoire **`/home/junon/d03/maitres/dupont/JAVA`** contient les paquetages de l'utilisateur DUPONT. Le répertoire **`.`** signifie le répertoire courant.

Bibliothèque

Java offre une bibliothèque standardisée, sous forme de paquetages thématiques qui couvrent la plupart des sujets d'intérêts actuels :

`java.lang` : classes de base du langage (chaînes, math, processus, exceptions,...),
`java.util` : structures de données (vecteurs, piles, tables, parcours,...),
`java.io` : entrées-sorties classiques (texte sur clavier-écran, fichiers,...),
`java.awt` : interfaces homme-machine(fenêtrage, événements, graphique, ...),
`java.net` : communications Internet (manipulation d'URL, de sockets,...),
`java.applet` : insertion de programmes dans des documents HTML

3 Types et structures de contrôle

3.1 Types en Java

En Java il y a une nette distinction entre les ***types primitifs*** (**`int`**, **`char`**, **`boolean`** ...) et les ***types construits*** (types classe ou tableau) : seuls les types primitifs ont la notion de "valeur" offerte par le langage, alors que les types construits n'offrent que des objets, nécessairement manipulés par référence.

3.1.1 Types primitifs

`char`	caractères (Unicode, sur 16 bits, surensemble des caractères ASCII)
`byte`, **`short`**, **`int`**, **`long`**	nombres entiers 8, 16, 32 et 64 bits
`boolean`	booléens (valeurs **`true`** et **`false`**)
`float, double`	nombres flottants, simple et double précision

opérateurs usuels :

`+`, **`-`**, **`*`**, **`/`**, **`%`** (modulo)	opérateurs arithmétiques pour entiers et flottants.
`==`, **`!=`**	égalité et non-égalité, définis pour tout type primitif.
>, >=, <, <=	comparaisons arithmétiques pour entiers et flottants.
`!`, **`&`**, **`|`**, **`&&`**, **`||`**	non, et, ou, et conditionnel, ou conditionnel, pour les booléens (conditionnel : n'évalue pas le second opérande si le premier détermine le résultat).

© IFSIC - Université de Rennes 1 - 2003

3.1.2 Types énumérés

Java (à partir de la version 1.5) offre les types énumérés. Un tel type possède un ensemble fini de valeurs désignées par autant d'identificateurs. On peut par exemple définir le type **`Orientation`** comme un type énuméré possédant 4 valeurs notées **`nord`**, **`sud`**, **`est`** et **`ouest`** :

```
enum Orientation {nord,sud,est,ouest};
```

Les déclarations des données d'un type énuméré utilisent le nom du type, et les désignations de valeurs se font au moyen de la notation ***nomDuType*`.`*identificateur***. Le seul opérateur défini sur un type énuméré est le test d'égalité, noté "**`==`**". Exemple :

`Orientation o;` déclaration de variable

```
...
if (o==Orientation.nord) {...}
```
test d'égalité

Une expression de type énuméré peut servir d'argument d'un aiguillage. Dans ce cas les valeurs de type énuméré qui qualifient les branches de l'aiguillage sont notées sans le nom du type en préfixe, car le type de l'argument de l'aiguillage lève toute ambiguïté :

```
switch (o) {
case nord : ... break;
case est  : ... break;
case sud  : ... break;
case ouest: ... break;
}
```

L'usage des types énumérés remplace avantageusement l'ancienne technique qui consistait à définir des constantes entières, car ainsi le compilateur peut effectuer un contrôle de type précis.

3.1.3 Types classes et tableaux

Les tableaux et les objets de type classe présentent les points communs suivants : ils sont toujours manipulés par ***référence***, et leur déclaration ne fait qu'associer l'identificateur à une variable de nature référence initialisée à **`null`**. Pour que cet identificateur désigne un objet, il faut lui affecter soit le résultat d'une création, grâce à la primitive **`new`**, soit un objet déjà existant.

La déclaration d'un identificateur de type classe a déjà été vue. Un tableau se déclare ainsi :

`int[] T;` tableau d'entiers
`Robot[] R;` tableau d'éléments de type **`Robot`**.

Une déclaration de tableau n'indique pas de taille. La taille d'un tableau est fournie au moment de sa création, de la façon suivante :

`T=new int[20];` crée un tableau de 20 entiers, référencé par **`T`**
`R=new Robot[5];` crée un tableau de 5 objets de type **`Robot`**, référencé par **`R`**

Les indices des tableaux commencent à 0.

© IFSIC - Université de Rennes 1 - 2005

L'accès à un élément se note ainsi : **`T[15]`**
On peut connaître la taille d'un tableau **`T`** par l'attribut **`T.length`**.

Java n'offre pas de notion spécifique pour les tableaux à plusieurs dimensions : on les réalise simplement comme des tableaux de tableaux.
`int[][] M;` tableau de tableaux d'entiers.

Gestion automatique de la mémoire : ramasse-miettes

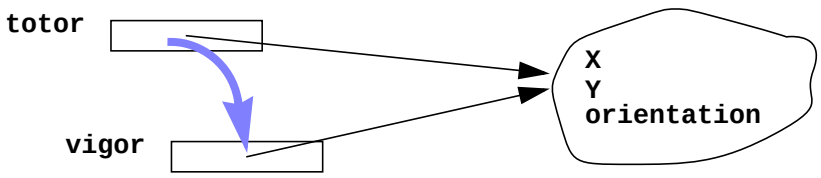
En Java, les objets construits (de type classe ou tableau) sont créés très dynamiquement, par exécution de **`new`**. Il sont créés dans le tas, zone mémoire gérée par le langage, et leur place est automatiquement récupérable lorsqu'ils ne sont plus référencés. Cette récupération est effectuée par un ramasse-miettes (*garbage collector*) qui, selon les mises en œuvre, est déclenché lorsqu'il n'y a plus de place ou bien fonctionne plus ou moins en parallèle avec l'exécution des programmes.

3.2 Les références

Les identificateurs de type classe ou tableau, que ce soient des identificateurs de variables ou de paramètres de méthode, désignent toujours un objet à travers une référence. L'exemple suivant illustre ce que cela induit :

```
class TestReferences {
public static void main(String[] z) {
  Robot totor = new Robot(20,30,Robot.Est);
  Robot vigor;
  System.out.println(totor.X);      imprime 20
  vigor = totor;
  vigor.avancer();
  System.out.println(totor.X);          imprime 21
}}
```

Un objet de type **`Robot`** est créé et sa référence est captée par l'identificateur **`totor`**. Après l'affectation **`vigor=totor`**, l'identificateur **`vigor`** capte la référence associée à **`totor`**, et donc **`vigor`** désigne le même objet. Cela se voit à l'exécution : lorsque l'on fait avancer **`vigor`**, cela modifie la position de **`totor`**, puisque c'est le même objet.



Tests d'égalité

L'opérateur de test d'égalité (**`==`**) existe pour les expressions de type classe ou tableau, mais il signifie la comparaison des références et non celle des valeurs des objets. Il permet donc de savoir si deux expressions désignent le même objet.
Si on a besoin de tester l'égalité des valeurs, il faut la programmer. Par convention, on le fait au moyen d'une méthode que l'on appelle **`equals`**.

© IFSIC - Université de Rennes 1 - 2003

```
class TestEgalite {
public static void main(String[] x) {
  Robot totor = new Robot(20,30,Robot.Nord);
  Robot kador = new Robot(20,30,Robot.Nord);
  Robot vigor;
  vigor = totor;
  if (vigor==totor) {System.out.println("vigor==totor");}
  else              {System.out.println("vigor!=totor");}
  if (kador==totor) {System.out.println("kador==totor");}
  else              {System.out.println("kador!=totor");}
  if (kador.equals(totor))
          {System.out.println("kador equals totor");}
  else    {System.out.println("kador non equals totor");}
}
}
```

Dans l'exemple ci-dessus, le test **`vigor==totor`** est ***vrai***, car **`vigor`** désigne le même objet que **`totor`**. En revanche, **`kador==totor`** est ***faux***. L'égalité des valeurs des objets référencés par **`kador`** et **`totor`** peut être testée par une méthode **`equals`** de la classe **`Robot`** :

```
boolean equals(Robot r) {
  return X==r.X && Y==r.Y && orientation==r.orientation;
}
```

Le même mécanisme s'applique aux tableaux : l'opérateur **`==`** entre tableaux teste si deux tableaux sont le même objet.
De façon similaire, l'opérateur **`!=`** entre expressions de type classe ou tableau teste si ces expressions désignent des objets différents.

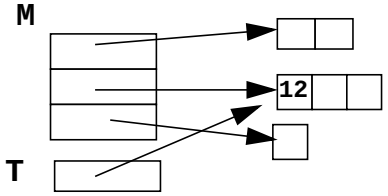
Tableaux à plusieurs dimensions

Le fait que Java considère les tableaux à plusieurs dimensions comme des tableaux de tableaux implique d'une part que les sous-tableaux peuvent être de tailles différentes, par exemple :

`int[][] M;`
`M= new int[3][];` crée un tableau de 3 tableaux d'entiers
`M[0]= new int[2];` crée le premier vecteur de **`M`** de taille 2
`M[1]= new int[3];` crée le deuxième vecteur de **`M`** de taille 3
`M[2]= new int[1];` crée le troisième vecteur de **`M`** de taille 1

et d'autre part que plusieurs tableaux peuvent se partager le même sous-tableau, ce qui peut induire des effets de bords dont il faut se méfier, par exemple :

`T=M[1];` **`T`** capte le deuxième vecteur de **`M`**
`T[0]=12;` ceci donne la valeur 12 à **`T[0]`** mais aussi à **`M[1][0]`**.



© IFSIC - Université de Rennes 1 - 2005

On peut toutefois directement créer une matrice, c'est-à-dire un tableau de tableaux de même taille par :

`M= new int[3][3];` crée une matrice 3 x 3

3.3 Chaînes de caractères : `String` et `StringBuffer`

Java offre les chaînes de caractères sous forme de deux classes **`String`** et **`StringBuffer`**. Ces classes sont riches en fonctionnalités.

La classe **`String`** offre des objets chaînes ***constants***. Ceci signifie que si l'on déclare par exemple :

```
String ch1 = new String("bonjour");
```

l'objet créé, maintenant désigné par **`ch1`**, ne peut être modifié, il vaudra toujours **`"bonjour"`**. Cela ne signifie nullement que **`ch1`** soit une constante. C'est ici une variable, capable de capter un autre objet, par exemple en l'affectant :

```
String ch2 = new String("au revoir");
ch1 = ch2;
```

`ch1` désigne alors la chaîne **`"au revoir"`**.

La forme **`new String("blabla")`** pour créer une chaîne de valeur **`"blabla"`** peut être abrégée simplement en **`"blabla"`**. Ainsi on peut déclarer directement :

```
String ch1 = "bonjour";
```

ou encore utiliser directement la notation de chaîne en paramètre effectif :

```
System.out.println("bonjour");
```

Voici quelques méthodes utiles de la classe **`String`** :

`static String valueOf(int i);` chaîne qui représente l'entier **`i`**
`static String valueOf(boolean b);` chaîne qui représente le booléen **`b`**
`static String valueOf(t x);` chaîne qui représente ***x*** de type primitif ***t***

Les méthodes ci-dessus ne portent sur aucun objet (mot clé **`static`**). Pour les utiliser, on les préfixe par le nom de la classe :

`String.valueOf(12)` rend la chaîne **`"12"`**.

`boolean equals(String s);` teste l'égalité entre **`this`** et **`s`**
`String concat(String s);` rend la concaténation de **`this`** et **`s`**
`int length();` rend la longueur de **`this`**
`int indexOf(int c);` position de la première occurrence du caractère de code ASCII **`c`**, **`-1`** si le caractère n'apparaît pas
`char charAt(int i);` caractère en position **`i`** (numérotation à partir de 0)
`int compareTo(String s);` rend un entier <0, 0 ou >0 selon que **`this`** est inférieur, égal ou supérieur à **`s`** selon l'ordre alphabétique

La concaténation de chaînes est également offerte sous forme d'un opérateur fonctionnel infixe noté **`+`** : **`s1+s2`** : concaténation de **`s1`** et **`s2`**
Un opérande de l'opération **`+`** peut également être un caractère (il est converti en chaîne de taille 1) ou un nombre (il est converti en une chaîne qui est sa notation décimale).

© IFSIC - Université de Rennes 1 - 2003

La classe **`StringBuffer`** offre des objets chaînes ***modifiables***, c'est à dire des objets dont on peut changer des morceaux, dont on peut changer la taille, ...
Le principal usage de cette classe est la construction progressive et efficace de chaînes.

Constructeurs :

`StringBuffer();` chaîne vide
`StringBuffer(int length);` chaîne de taille **`length`**, non initialisée
`StringBuffer(String s);` chaîne initialisée avec la valeur de la **`String s`**

Exemples de méthodes :

`StringBuffer append(String s);`
ajoute les caractères de **`s`** en fin de **`this`**

`StringBuffer append(char c);`
ajoute le caractère **`c`** en fin de **`this`**

`String toString();`
chaîne constante ayant pour valeur la valeur courante de **`this`**

`int length();` longueur de **`this`**

Remarque : **`append`** agit sur **`this`**, mais de plus il rend **`this`** en résultat. C'est pourquoi son type est **`StringBuffer`** et non pas **`void`**.

L'exemple suivant illustre l'usage de **`StringBuffer`**. La classe **`Lecture`** offre la méthode **`chaine(String delimiteurs)`** pour la lecture de chaînes délimitées par un des délimiteurs donnés en paramètre :

```
class Lecture {
...
public static String chaine(String delimiteurs) {
  // lecture d'une chaine comprise entre delimiteurs
  StringBuffer b = new  StringBuffer();
  char c=unCar(); // lecture d'un caractere
  // ignore les delimiteurs de tete
  while (delimiteurs.indexOf(c)!=-1) {c=unCar();};
  // lit jusqu'au prochain delimiteur
  while (delimiteurs.indexOf(c)==-1)
  {b.append(c); c=unCar();};
  return b.toString();
}
}


class TestStringBuffer {
public static void main(String[] x) {
  String s;
  while(!(s=Lecture.chaine(" ,\r\n")).equals("fin")) {
      System.out.println(s);
  }
}}
```


© IFSIC - Université de Rennes 1 - 2005

3.4 Données constantes - `final`

Les constantes sont déclarées comme des composants de classe, en les qualifiant par l'attribut **`final`**. L'attribut **`final`** indique que l'identificateur ne peut apparaître en partie gauche d'affectation. Pour un composant de type primitif, ceci signifie que l'objet désigné possède une valeur constante. Pour un composant de type classe ou tableau cela signifie seulement que la référence associée à l'identificateur est constante. Cela n'empêche pas l'objet désigné de subir des modifications.

Exemples :
`final int nombreDeNains=7;`
`nombreDeNains` vaut en permanence 7.
On ne peut pas écrire : **`nombreDeNains=…`** ni **`nombreDeNains++`**, ...

`final Robot totor= new Robot();`
`totor` désigne en permanence le même objet de type **`Robot`**, celui créé à la déclaration. On ne peut pas écrire **`totor=...`**. En revanche, cet objet peut subir des modifications, par exemple on peut écrire **`totor.X=12`** ou encore **`totor.avancer()`**.

La valeur d'une constante n'est attribuée qu'une fois au cours de l'exécution, et en des endroits fixés par le langage :

- soit à l'endroit de sa déclaration,
- soit dans les constructeurs de la classe où elle est définie.

Cette seconde possibilité permet, par le paramètrage des constructeurs, d'attribuer une valeur différente à un même identificateur de constante pour des objets différents.
Si on a besoin de constantes non attachées à des objets, on les déclare avec l'attribut **`static`**.
L'exemple suivant illustre ces diverses possibilités :

```
class Robot {
static final int Nord = 1; static final int Est = 2;
static final int Sud = 3;  static final int Ouest = 4;
int X; int Y; int orientation;
final int Xorig; final int Yorig;
final Robot collegue;
Robot(int x, int y, int o, Robot c) {
X=x; Y=y; orientation=o; Xorig=x; Yorig=y; collegue=c;
}
Robot(int x, int y, int o) {
X=x; Y=y; orientation=o; Xorig=x; Yorig=y; collegue=null;
}

...
void retourCaseDepart() {
     X=Xorig; Y=Yorig;
}

void rejoindreCollegue() {
    if (collegue!=null) {X=collegue.X; Y=collegue.Y;}
}}
```


© IFSIC - Université de Rennes 1 - 2003

Les constantes **Nord**, **Sud**, **Est**, **Ouest** ne sont pas associées aux objets, et sont donc déclarées **static**. Cette version de **Robot** offre des composants constants associés à chaque objet : la position initiale et un partenaire appelé **collegue**. Ces composants constants sont initialisés dans les constructeurs, de façon à être dépendants des paramètres de création. Voici un exemple d'utilisation de cette classe :

```
class TestConstantes {
public static void main(String[] x) {
  Robot totor = new Robot(20,30,Robot.Nord);
  Robot vigor = new Robot(10,15,Robot.Sud,totor);
  totor.avancer();
  System.out.println("totor "+totor.X+" "+totor.Y);
  totor.retourCaseDepart();
  System.out.println("totor "+totor.X+" "+totor.Y);
  System.out.println("vigor "+vigor.X+" "+vigor.Y);
  vigor.rejoindreCollegue();
  System.out.println("vigor "+vigor.X+" "+vigor.Y);
}
}
```


© IFSIC - Université de Rennes 1 - 2005

3.5 Instructions

Pour rédiger les méthodes, on dispose, entre autres, des instructions traditionnelles : affectation de variable, appel de méthode et instructions construites au moyen des structures de contrôle usuelles telles que conditionnelle, boucle, cas.

3.5.1 Affectation

Forme générale : **`variable = expression;`**

évalue l'expression et donne sa valeur à la variable.

La variable peut être simplement désignée par un identificateur :
`i=i+1 ;` incrémente **`i`**

Elle peut également être désignée par une expression plus ou moins compliquée :

`M[2][6]=12;` range 12 dans l'élément 2,6 de la matrice **`M`**
`Robot.leMeilleur(totor,vigor).X = 12;`
affecte le champ **`X`** de l'objet **`Robot`** rendu en résultat par la méthode **`leMeilleur`**.

En java, comme en C et C++, certaines opérations d'affectation d'usage fréquent, comme les incrémentations ou les décrémentations, admettent une notation abrégée :

`++i`	réalise **`i=i+1`** et rend la valeur de **`i`** après incrémentation
`--i`	réalise **`i=i-1`** et rend la valeur de **`i`** après décrémentation
`i++`	réalise **`i=i+1`** et rend la valeur de **`i`** avant incrémentation
`i--`	réalise **`i=i-1`** et rend la valeur de **`i`** avant décrémentation

3.5.2 Conditionnelle

`if (`*cond*`) {`*instructions*`}`

Exécute ***instructions*** si ***cond*** est vraie.

`if (`*cond*`) {`*instructions*$_1$`} else {`*instructions*$_2$`}`

Exécute ***instructions***$_1$ si ***cond*** est vraie, exécute ***instructions***$_2$ si ***cond*** est fausse.

Une forme plus générale est :

`if (`*cond*$_1$`) {`*instructions*$_1$`}`
`else if (`*cond*$_2$`) {`*instructions*$_2$`}`
`...`
`else if (`*cond*$_k$`) {`*instructions*$_k$`}`
`else {`*instructions*`}`

Exécute le bloc d'instructions correspondant à la première condition vraie.

© IFSIC - Université de Rennes 1 - 2003

3.5.3 Boucle

Boucle “tantque”

`while (`*cond*`) {`*corps*`}`

Exécute les instructions de ***corps*** tant que ***cond*** est vraie.

Boucle “pour”

`for (`*init*`; `*cond*`; `*progression*`) {`*corps*`}`

Cette forme de boucle est équivalente à

***init*`; while (`*cond*`) {`*corps progression*`;}`**

Dans une utilisation saine, ***init*** est une instruction qui initialise les variables qui contrôlent l’itération, ***cond*** est la condition de poursuite, et ***progression*** est une instruction qui fait évoluer les variables de contrôle.

On peut de plus déclarer une variable dans la partie ***init***, ce qui crée la variable de contrôle de boucle juste pour cette itération.

Exemple : calcul de la somme des éléments d’un tableau **`T[0]`**...**`T[9]`**

```
int s=0;
for (int i=0; i<10; i=i+1) {s=s+T[i];}
```

3.5.4 Instruction cas

```
switch (expr) {
case v1 : instructions1 break;
case v2 : instructions2 break;
...
default : instructions break;
}
```

expr est une expression de type **`byte`**, **`char`**, **`short`**, **`int`** ou **`long`** et les v_i sont des valeurs constantes du même type que ***expr***. Cette construction exécute le bloc d’instructions correspondant à la première valeur v_i égale à ***expr***.

La branche **`default`** est optionnelle et est exécutée si aucun des v_i n’est égal à ***expr***. On peut regrouper plusieurs valeurs qui nécessitent le même traitement :

```
case u : v : w : instructions break;
```


© IFSIC - Université de Rennes 1 - 2005

3.6 Entrées/sorties

Java offre deux sortes de communications avec l'extérieur :

- des entrées-sorties traditionnelles : lectures et écritures de textes sur clavier/ écran ou sur fichiers,
- des interactions à travers un système de fenêtrage : création de fenêtres, création de boîtes de dialogues, réaction à des événements.

Le système de fenêtrage est présenté au chapitre 7, page 52.
Pour les entrées-sorties écran/clavier, on dispose des appels de méthodes suivants :

`System.out.print(String s);` impression de la chaîne **`s`**
`System.out.println(String s);` idem avec retour à la ligne

`System.in.read();` lecture d'un caractère

Ce sont des méthodes des classes **`PrintStream`** et **`InputStream`** définies dans le paquetage **`java.io`**. Leur profil est :

```
void println(String s);
void print(String s);
int System.in.read() throws IOException;
```

`System` est une classe du paquetage **`java.lang`**. Cette classe définit l'objet **`out`** de classe **`PrintStream`** pour les impressions sur écran et l'objet **`in`** de classe **`InputStream`** pour les lectures au clavier.

La méthode **`read()`** nécessite quelques explications : elle lit un caractère, mais elle rend un **`int`** qui est le code ASCII du caractère frappé (c'est curieux, mais c'est ainsi). Pour obtenir le caractère Java officiel correspondant, de type **`char`**, il faut pratiquer une conversion (un ***cast***) qui se note ainsi :

```
char c; ... c = (char) System.in.read();
```

En outre, cette méthode est susceptible de déclencher une ***exception*** (voir paragraphe suivant) dans le cas où la lecture se passe mal (cela ne peut pas se produire pour une lecture clavier, mais **`read()`** est plus générale). Le mécanisme des exceptions est décrit au paragraphe suivant. Ceci oblige à utiliser cette méthode au sein d'un bloc **`try`** qui prévoit un éventuel traitement d'exception :

```
try { c = (char) System.in.read(); }
catch(IOException e) {c= '#';};
```

Dans cet exemple, le traitement d'exception affecte le caractère **`'#'`** à **`c`**.

Pour l'interprétation numérique de chaînes de chiffres, la classe **`Integer`** offre la méthode statique suivante :

`int parseInt(String s) throws NumberFormatException`
qui rend l'entier représenté en décimal par la chaîne **`s`**.

Cette méthode déclenche une exception si **`s`** ne satisfait pas à la syntaxe de représentation décimale d'un entier.

© IFSIC - Université de Rennes 1 - 2003

Pour programmer les exercices, on utilisera avec profit les méthodes de la classe **`Lecture`** dont le source est donné ci-dessous.

```
  public static char unCar() {
  // effet : lit un caractère frappé
  // résultat : le caractère frappé
    char c;
    try { c = (char) System.in.read();}
    catch(IOException e) {c= (char) 0;};
    return c;
  }

  public static String chaine(String delimiteurs) {
  // prérequis : delimiteurs.length()!=0
  // effet : lit une suite de caractères
  // résultat : la chaine formée des caractères compris
  //entre les delimiteurs indiqués
    StringBuffer b = new  StringBuffer();
    char c=unCar();
    // ignore les delimiteurs de tete
    while (delimiteurs.indexOf(c)!=-1) {c=unCar();};
    // lit jusqu'au prochain delimiteur
    while (delimiteurs.indexOf(c)==-1) {
      b.append(c); c=unCar();
    }
    return b.toString();
  }

  public static int unEntier() {
  // effet : lit une chaîne de caractères comprise entre
  // les délimiteurs ' ', '\r' ou '\n'
  // résultat : l'entier representé en décimal par cette
  // chaîne
    String s=Lecture.chaine(" \r\n");
    try { return Integer.parseInt(s);}
    catch(NumberFormatException e) {
      System.err.println("\nErreur lecture d'entier");
      System.err.println("valeur 0 retournée");
      return 0;
    }
  }
}
```


© IFSIC - Université de Rennes 1 - 2005

3.7 Traitement d'exception : `throw-try-catch`

La notion d'exception permet de traiter de manière plus souple et plus lisible les cas exceptionnels, essentiellement les cas d'erreurs. Si on traite les cas exceptionnels comme des cas normaux, cela exige que les procédures rendent des résultats supplémentaires, qu'il faut tester au moyen d'instructions conditionnelles, ... Ceci alourdit la programmation au point que la logique de traitement des cas normaux se trouve noyée dans celle des cas exceptionnels.
En Java, les cas exceptionnels peuvent donner lieu à un ***déclenchement d'exception***. Une exception peut être déclenchée par l'interpréteur du langage, par exemple lors d'une division par 0 ou d'un accès hors des bornes d'un tableau.

On peut aussi en déclencher par une instruction :

```
throw(e);
```

où **`e`** est un objet de classe **`Throwable`**, structure de donnée qui contient des informations concernant la nature de l'exception. Pour chaque catégorie d'exception, il existe une classe dérivée de **`Throwable`** (voir plus loin, chapitre sur l'héritage), dont le nom est de la forme **`xxxError`** ou **`xxxException`**, par exemple :

`IOException` : exception liée aux entrées/sorties
`NullPointerException` : accès à travers **`null`**
`IndexOutOfBoundsException` : accès hors des bornes d'un tableau
`NumberFormatException` : syntaxe incorrecte de représentation d'un nombre par une chaîne de caractères.
...

Si une méthode est susceptible de déclencher une exception, cela doit être indiqué dans son profil, au moyen de la directive **`throws`** :

```
int ppp() throws xxxException { ... }
```

Si une méthode **`p()`** utilise une méthode **`q()`** susceptible de déclencher une exception, il faut :

- soit capter, et éventuellement traiter, l'exception au moyen d'un bloc **`try`** au sein de la méthode **`p()`**,

```
void p() { ...
  try { ... q() ...}
  catch(xxxException e){ traitement de e }
}
```

- soit propager l'exception, en indiquant au moyen de la directive **`throws`** que **`p()`** est elle-même susceptible de déclencher cette exception :
  `void p() throws xxxException { ... q() ... }`

© IFSIC - Université de Rennes 1 - 2003

Exercice 1

On considère la pile d'entiers dotée de moyens de parcours. Programmer une nouvelle version en réalisant la pile par chaînage de maillons. Définir deux initialisations pour la classe de parcours :

- initialisation sur le sommet de la pile,
- initialisation avec l'état d'un parcours passé en paramètre.

Rédiger un programme qui :

- crée une pile et y empile 7, 5, 4, 6, 5, 2, 3
- utilise les moyens de parcours pour chercher la première valeur qui figure en double à partir du sommet (5 dans l'exemple).

Facultatif mais amusant :
pour un parcours p1 initialisé avec l'état d'un parcours p2, on peut vouloir détecter la situation absurde ou p2 n'est pas un parcours sur la même pile que p1. Chercher le moyen de détecter cette situation.
Pour citer l'instance courante de la classe englobante, la syntaxe est :
***classeEnglobante*`.this`**

Exercice 2

On se propose de réaliser une structure de données ***table de correspondance*** permettant d'associer des ***valeurs*** à des ***clés***. Les clés seront des chaînes de caractères et les valeurs des nombres entiers. Ceci permet par exemple de représenter le résultat d'une course en prenant pour clés les noms des coureurs et pour valeurs les temps réalisés :

`<("`dupont**`",28), ("alfred",45), ("durand",12)>`**

La classe **`TableDeCorrespondance`** doit offrir les fonctionnalités suivantes :

- Constructeur permettant de créer une table vide.
- Ajout d'une association **`(`*clé*`,`*valeur*`)`** à la table. Si la clé se trouve déjà dans la table, cela remplace l'ancienne valeur associée par la nouvelle.
- Retrait d'une association indiquée par sa clé. Ne fait rien si la clé est absente.
- Recherche de la valeur associée à une clé. Si la clé est absente, cela doit générer une exception de type **`CleAbsenteException`**.

On commencera par rédiger les ***spécifications*** de **`TableDeCorrespondance`** : données, constructeur et méthodes publiques avec commentaires indiquant leur rôle.

Mise en œuvre :

On décide de représenter une table de correspondance au moyen d'un tableau **`T`** de paires **`(`*clé*`,`*valeur*`)`** et d'une variable entière **`nbElements`** qui indique le nombre d'associations présentes dans la table. Les paires seront réalisées au moyen d'une classe interne **`Paire`**. La taille du tableau, constante statique, fixera la capacité maximum de la table. Un dépassement de capacité génèrera une exception de type **`SaturationException`**.
Conseil : rédiger une fonction auxiliaire : **`int indiceDeCle(String c)`** qui rend l'indice de l'association de clé **`c`** si elle existe, ou **`nbElements`** si la clé **`c`** est absente.

© IFSIC - Université de Rennes 1 - 2005

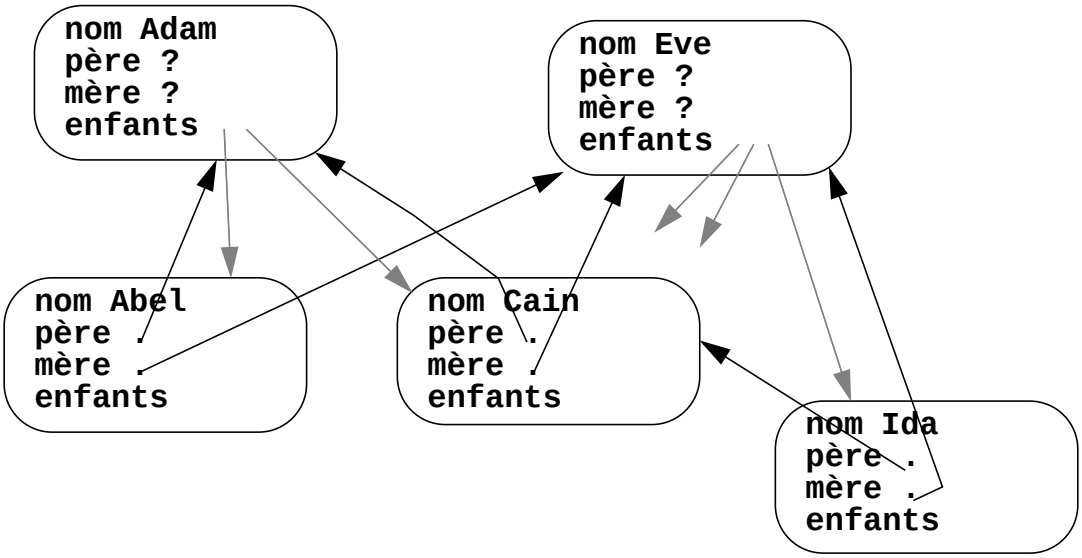
Test :

Rédiger un programme principal qui construit, par ajouts successifs, la table suivante :
`<("dupont",28), ("alfred",45), ("durand",12)>`
puis qui recherche et imprime les valeurs associées à “durand” puis à “toto”.

Exercice 3

On désire construire des arbres généalogiques de personnes.

Rédiger la classe **`Personne`** qui satisfait aux spécifications suivantes :
Une personne possède un nom, un père, une mère et des enfants.
Lors de la création d’une personne, on indique son nom, son père et sa mère.
La liste de ses enfants est initialisée à vide et cette personne est rajoutée dans la liste des enfants de son père et de sa mère.



On utilisera la classe **`ListePersonnes`**, liste d’objets de type **`Personne`**, dotée de moyens de parcours :

`ListePersonnes()` : constructeur liste vide
`void ajouteEnQueue(Personne p)` : ajout d’un élément en queue de liste,
`ListePersonnes.Parcours nouveauParcours()` :
rend un parcours initialisé en tête de la liste

méthodes de la classe **`Parcours`** :
`void suivant()` : fait avancer le parcours d’une position
`Personne elementCourant()` : élément courant du parcours
`boolean estEnFin()` : test de fin de parcours

Rédiger un programme principal qui crée la population illustrée sur la figure, puis imprime la liste des enfants de la grand-mère paternelle de **`Ida`**.

© IFSIC - Université de Rennes 1 - 2003

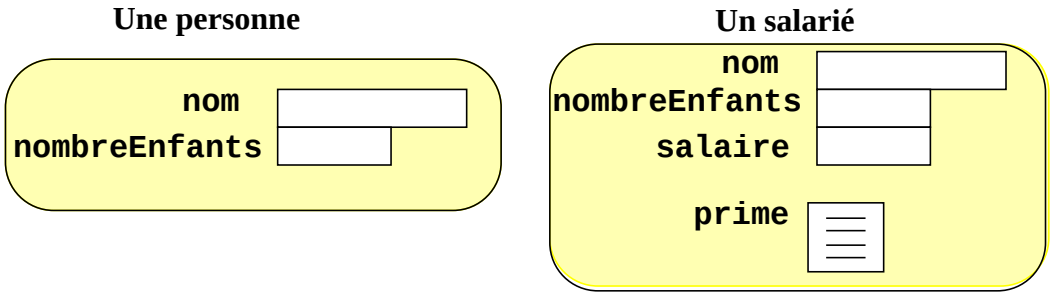
4 Héritage

4.1 Usage simple de l'héritage : ajout de propriétés

On a parfois besoin de définir un type d'objet similaire à un type existant, avec quelques propriétés supplémentaires. L'héritage permet de définir ce nouveau type sans tout reprogrammer : il suffit de déclarer que le nouveau type hérite du précédent et on se borne à rédiger ses fonctionnalités supplémentaires. Ceci constitue l'utilisation la plus simple de l'héritage. Par exemple, on peut dériver un type **`Salarié`** à partir d'un type plus général **`Personne`** :

```
class Personne {
String nom;
int nombreEnfants;
Personne(String n) { nom=n; nombreEnfants=0;}
}

class Salarie extends Personne {
int salaire;
int prime() { return 5*salaire*nombreEnfants/100;}
Salarie (String n, int s) {super(n); salaire=s;};
}
```

La notation **`class Salarié extends Personne {...}`** indique que la classe **`Salarié`** ***hérite*** de la classe **`Personne`**. On dit également que **`Salarié`** est une classe ***dérivée*** de **`Personne`**. **`Personne`** est une ***classe de base*** de **`Salarié`**. Le type **`Salarié`** possède toutes les propriétés du type **`Personne`**, mêmes composants (données et méthodes), plus quelques nouvelles, le champ **`salaire`** et la fonction **`prime()`**.



Au début du corps d'un constructeur de la classe dérivée, on peut appeler explicitement un constructeur de la classe de base avec les paramètres souhaités grâce à la notation :

`super(`*paramètres***`)`**

Si on omet cet appel, l'exécution du constructeur est de toute façon précédée par l'exécution du constructeur sans paramètre de la classe de base.
Si une classe n'a aucun constructeur explicitement défini, Java définit le constructeur par défaut, qui est sans paramètre, et qui, pour une classe héritière, consiste en l'appel du constructeur sans paramètre de la classe de base.

© IFSIC - Université de Rennes 1 - 2005

4.1.1 Accessibilité : `public`, `protected`, `private`

En ce qui concerne l'accessibilité des composants d'une classe de base à partir des textes des classes dérivées, le langage offre les quatre modes suivants :

aucun attribut : accessibles par les classes qui font partie du même paquetage, inaccessibles par les autres.

`public` : accessibles par toutes les classes

`protected` : accessibles par toutes les classes ***dérivées***, et les classes du même paquetage, inaccessibles par les autres

`private` : inaccessibles par toutes les classes

L'attribut **`protected`** permet de rendre accessibles certains membres pour la conception d'une classe dérivée mais inaccessibles pour les utilisateurs de la classe de base. Comme le montre l'exemple suivant, l'accès à un composant **`protected`** est interdit en situation d'***utilisation*** de la classe de base : depuis la classe **B**, bien qu'héritière de **A**, l'accès **`a.JJ`** est interdit, car il s'agit d'une utilisation de **A**.

```
package aa;

public class A {
...
protected int JJ;
...
}
```

```
package bb;
import aa.*;

class B extends A {
...
void PP() {          autorisé
... JJ++; ... B b; ... b.JJ++;
...
A a; ... a.JJ++; ...
}          interdit
}

class C {
...
void QQ() {A a; ... a.JJ++; ...}
}                          interdit
```

4.2 Compatibilité entre types

Le fait qu'un type **B** hérite d'un type **A** signifie que **B** est un ***sous-type*** de **A**, c'est-à-dire qu'un objet de type **B** ***est*** également de type **A**. Par exemple, un **`Salarié`** est également une **`Personne`**. Donc toute opération applicable au type **A** est également applicable au type **B**. Le type **B** est dit ***plus précis*** que **A**, et **A** ***plus vague*** que **B**.

La règle générale de compatibilité de types peut être informellement énoncée ainsi : partout où une expression de type **A** est attendue, une expression de type plus précis que **A** est acceptable. En revanche, l'inverse est interdit. Cette règle s'applique essentiellement en deux circonstances : en partie droite d'une affectation ou bien en tant que paramètre effectif de procédure. Par exemple, avec les déclarations suivantes :

```
Personne p ...;     Salarié s ...;
void Q(Personne p) { ... };
```

ces instructions sont permises : **`p = s;  Q(s);`**

© IFSIC - Université de Rennes 1 - 2003

Pendant l'exécution, la variable **p** et le paramètre formel **p** sont des références sur l'objet précis de type **Salarie** qui leur est assigné.
Cependant **p** ne donne pas directement accès aux membres du type précis **Salarie** : on ne peut pas écrire **p.salaire**, ni **p.prime()**. Le compilateur refuse ces expressions, à juste titre car l'objet désigné par **p** pourrait être parfois du type **Personne**, et ces accès n'auraient aucune signification.

Pour profiter vraiment du fait que l'objet désigné par **p** est du type précis **Salarie**, il faut utiliser la notion de ***méthode virtuelle*** décrite dans un paragraphe ultérieur.

4.3 Classe `Object`

Java offre une classe **Object** dont héritent implicitement toutes les classes programmées en Java. Cette classe définit quelques fonctionnalités de base, par exemple :

```
public class Object {
  public Object();
  public boolean equals(Object o); // test d'égalité
  public String toString(); // représentation imprimable
  ...
}
```

Ces méthodes sont des méthodes virtuelles dont les fonctionnalités sont destinées à être redéfinies dans les classes particulières (voir paragraphe sur les méthodes virtuelles). Si on ne les redéfinit pas, elles réalisent une fonctionnalité par défaut :

- **equals** indique si **this** et **o** désignent le même objet,
- **toString** rend en résultat la chaîne de caractères constituée du nom de la classe et d'un numéro hexadécimal identifiant l'objet.

4.4 Test de type : `instanceof`

Le langage Java permet de tester le type exact d'un objet au moment de l'exécution, au moyen de la primitive **instanceof** :

expression **instanceof** ***classe***
rend **true** si l'expression désigne un objet de la classe indiquée, **false** sinon.

Ceci permet de faire jouer à l'héritage un rôle similaire à celui d'union de types : le type de base devient dans ce cas l'union des types dérivés. C'est un usage pratique mais un peu détourné de l'héritage.

4.5 Forçage de type (*cast*)

Le forçage de type (***cast***) est une expression de la forme : **(*type*)** ***expr***
Sa signification est différente pour les types primitifs et les références à objets.
Pour les ***types primitifs***, il s'agit d'une conversion "standard", par exemple :
(int) Math.PI est la conversion en entier de type **int** du **double** ***3.1415...***, qui de façon standard est sa partie entière, c'est-à-dire ***3***.
Pour les références aux objets, c'est la demande au compilateur d'accepter la référence comme désignant un objet du type indiqué. Exemple :

```
Personne p; Salarie s; ... s = (Salarie) p;
```


© IFSIC - Université de Rennes 1 - 2005

Il est de la responsabilité du programmeur de garantir que l'objet désigné est bien du type indiqué, sinon cela provoquera une erreur à l'exécution (erreur généralement difficile à détecter). Cette sorte de ***cast*** peut s'utiliser conjointement avec un test de type, auquel cas il n'y a pas de risque d'erreur si le test est cohérent :

```
if( p instanceof Salarie) {s = (Salarie) p; ... }
```

Exercice 4

Une application de gestion de bibliothèque doit manipuler des documents de natures disparates, par exemple des livres et des dictionnaires. Tous les documents ont un titre. Les autres attributs varient selon la catégorie du document : un livre possède un auteur et un nombre de pages, un dictionnaire est caractérisé par le nombre de définitions de mots qu'il contient. Bien que de natures diverses, les documents doivent pouvoir être manipulés de façon homogène en tant que simples documents, par exemple pour en constituer des listes. On définit pour cela les classes **`Document`**, **`Livre`** et **`Dictionnaire`**.

1 - Programmer ces classes.

2 - Indiquer les lignes du programme suivant qui constituent des erreurs de syntaxe :

```
class TestBibli {
  public static void main(String[] x) {
    Livre pereGoriot = new Livre(...);
    Document doc;  Livre livre;
    doc = pereGoriot;
    System.out.println(doc.titre);
    System.out.println(doc.auteur);
    livre = doc;
  }
}
```

3 - On dispose de la classe **`ListeDeDocuments`** :

`ListeDeDocuments()` : constructeur liste vide
`void ajouteEnQueue(Document p)` : ajout d'un élément en queue de liste,
`ListeDeDocuments.Parcours nouveauParcours()` :
rend un parcours initialisé en tête de la liste

méthodes de la classe **`Parcours`** :
`void suivant()` : fait avancer le parcours d'une position
`Document elementCourant()` : élément courant du parcours
`boolean estEnFin()` : test de fin de parcours

© IFSIC - Université de Rennes 1 - 2003

Rédiger la partie manquante du programme suivant qui doit compter le nombre de livres et le nombre de dictionnaires présents dans la bibliothèque.

```
class TestdeClasse {
public static void main(String[] x) {
ListeDeDocuments bibli= new ListeDeDocuments();

bibli.ajouteEnQueue(new Livre("Le pere Goriot","Balzac",458));
bibli.ajouteEnQueue(new Livre("Nounours", "Chantal Goya", 14));
bibli.ajouteEnQueue new Dictionnaire("Larousse", 4500));
bibli.ajouteEnQueue(new Livre("Tintin", "Herge", 62));
bibli.ajouteEnQueue(new Dictionnaire("Petit Robert", 5000));

int nbLivres=0; int nbDicos=0;

...

System.out.println("nombre de livres = "+nbLivres);
System.out.println("nombre de dictionnaires = "+nbDicos);
}
}
```


© IFSIC - Université de Rennes 1 - 2005

4.6 Méthodes virtuelles

4.6.1 Définition de méthodes virtuelles

Les ***méthodes virtuelles*** permettent de prévoir des opérations similaires sur des objets d'espèces différentes. On les appelle également ***méthodes différées*** pour insister sur le fait qu'elles sont destinées à être définies ou redéfinies pour chaque sous-espèce d'objets que l'on inventera ultérieurement.
En Java, toute méthode est potentiellement virtuelle, sauf si on lui donne l'attribut **`final`**, auquel cas elle ne pourra être redéfinie par les classes dérivées.

En Java, une classe peut annoncer une méthode sans la définir. Une telle classe est dite ***abstraite*** et elle doit être introduite par le mot clé **`abstract`**. Les méthodes qui ne sont pas définies sont également qualifiées **`abstract`** et seul leur profil est indiqué.

```
abstract class A {
...
abstract void P();
...
void Q() {...}
}
```

```
class B extends A {
...
void P(){...}
...
void Q() {...}
}
```

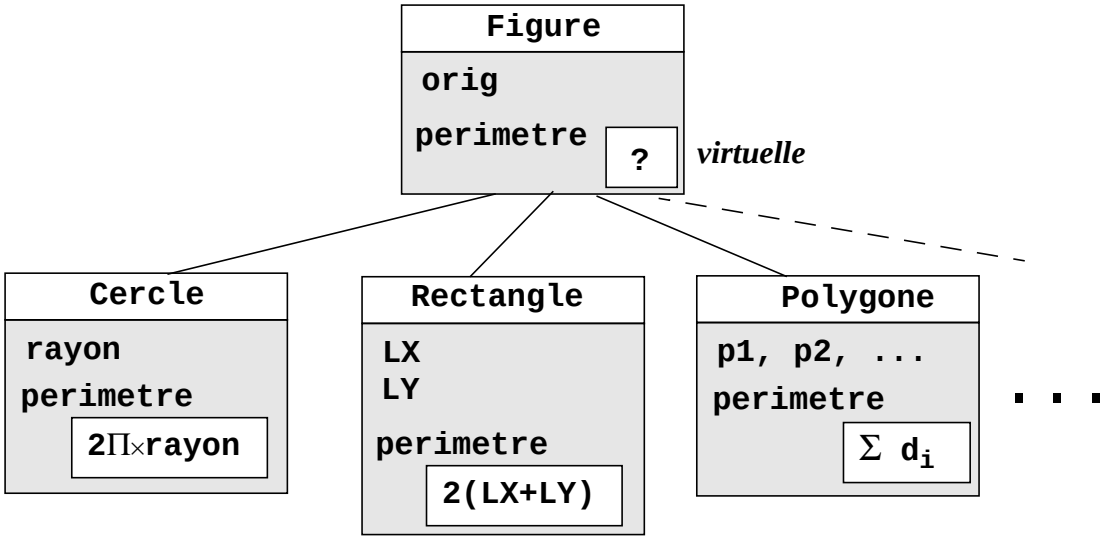
Dans l'exemple ci-dessus, la classe **A** annonce mais ne définit pas la méthode **P**. C'est une classe abstraite. La classe dérivée **B** définit **P** et redéfinit **Q**.
Une classe abstraite n'ayant pas toutes ses méthodes définies, il est interdit de créer des objets de ce type.
Comme exemple simple, on peut considérer diverses catégories de figures : les cercles, les rectangles, les polygones. Toutes ces variétés d'objets possèdent la notion de périmètre. Cependant la réalisation de l'opération est différente selon la catégorie. Ajoutons à cela que l'on ne connaît pas toutes les sortes de figures qui seront inventées par les programmeurs au moment où on conçoit le type général des figures. Pour cela, on indique, dans la classe **`Figure`**, que la fonction **`perimetre`** est abstraite.

```
abstract class Figure {
  Point orig; // point origine de la figure
  Figure(Point o) {orig=new Point(o);}
  abstract double perimetre();
}

class Point { double X; double Y;
  Point(double x, double y) {X=x; Y=y;}
  Point(Point o) {X=o.X; Y=o.Y;}
}

class Cercle extends Figure {
  private static final double pi=3.141592;
  double rayon;
  Cercle(Point centre, double r) {super(centre); rayon=r;}
  double perimetre() {return 2*pi*rayon;}
}
```


© IFSIC - Université de Rennes 1 - 2003

```
class Rectangle extends Figure {
  double LX; double LY; // hauteur, largeur
  Rectangle(Point coin,double lx,double ly) {
    super(coin); LX=lx; LY=ly;
  }
  double perimetre() {return 2*(LX+LY);}
}

class Polygone extends Figure {
  ...
}
```

Les diverses espèces de figures sont réalisées sous forme de classes héritières de **`Figure`** et chacune définit une version de la fonction **`perimetre()`** :



La classe générale **`Figure`** est ***abstraite***, et on ne peut pas créer d'objets de cette classe : sa seule raison d'être est de regrouper en une classe unique les diverses variétés de figures.

4.6.2 Liaison dynamique des méthodes virtuelles

Chaque fois qu'une méthode **`perimetre()`** est appelée sur un objet de type **`Cercle`**, **`Rectangle`** ou **`Polygone`**, c'est la méthode du type précis de l'objet qui est exécutée. Ceci a lieu même si la désignation de l'objet précis est de type vague. Le compilateur ne peut pas décider quelle méthode appeler ; seul un mécanisme dynamique peut appeler la bonne procédure au moment de l'exécution. C'est ce dernier point qui donne leur puissance aux méthodes virtuelles. Par exemple :

```
Cercle c= new Cercle(Point(4,10),12);
Rectangle r= new Rectangle(Point(10,110),20,30);
Figure f;
...
f = c; System.out.println(f.perimetre());  perimètre de cercle
f = r; System.out.print(f.perimetre());    perimètre de rectangle
```


© IFSIC - Université de Rennes 1 - 2005

Même si la méthode **perimetre** était définie au niveau de la classe **Figure**, c'est la méthode de la classe précise de l'objet, **Cercle** ou **Rectangle**, qui serait appelée. Au moment de l'appel **f.perimetre()**, la méthode convenable est appelée grâce à une information mémorisée dans chaque objet. Cette information n'est pas très coûteuse : c'est généralement un pointeur sur une description de la classe de l'objet.

Exercice 5
Indiquer les résultats affichés par le programme suivant :

```
class A { void p() {System.out.print(" A ");} }
class B extends A { void p() {System.out.print(" B ");} }

class TestLiaisonDynamique {

  static void q(A a) {a.p();}

  public static void main(String[] x) {
    A a= new A(); B b=new B();
    a.p(); q(a);
    a=b;
    a.p(); q(a); q(b);
  }
}
```

4.7 Quelques utilisations des méthodes virtuelles

Parmi les nombreuses utilisations possibles des méthodes virtuelles, on peut signaler les suivantes :

- manipulation de collections d'objets de types similaires mais distincts,
- rédaction de modules logiciels extensibles,
- utilisation de plusieurs représentations concrètes d'un même type abstrait.

Collection de données de types similaires

Comme on vient de le voir avec les diverses catégories de figures, les méthodes virtuelles permettent de manipuler de façon homogène des collections d'objets de types similaires mais distincts. Les diverses variétés (**Cercle**, **Rectangle**, ...) héritent d'un type vague unique représenté par une classe de base (**Figure**). Pour respecter les règles de typage usuelles, les collections de tels objets (listes, ensembles, tables, ...) sont gérées au moyen d'identificateurs déclarés du type vague. Ainsi, bien que l'on ignore le type précis des objets désignés, lorsqu'on appelle une méthode de la classe, la méthode du type précis de l'objet est automatiquement appelée.
Dans ce genre d'utilisation, jamais aucun objet de la classe de base n'est créé. La classe de base ne sert qu'à regrouper les diverses espèces. C'est un peu comme dans la nature, il n'existe pas d'animal "mammifère", un animal est toujours d'une espèce précise, "vache", "chien", "chat"... La classe de base est alors qualifiée d'***abstraite***. Il est préférable de ne pas y définir les méthodes virtuelles, en utilisant l'attribut **abstract**. Ainsi le compilateur refusera la création d'objets de ce type.

© IFSIC - Université de Rennes 1 - 2003

Modules logiciels extensibles

Les méthodes virtuelles facilitent la rédaction de logiciels extensibles, c'est-à-dire qui peuvent être modifiés et enrichis ultérieurement. L'héritage et les méthodes virtuelles permettent deux formes d'extensions :

- **Modification de fonctionnalité** : étant donné une classe **`T1`** dotée de méthodes virtuelles, un programmeur peut ultérieurement en dériver par héritage une classe **`T2`** mieux adaptée à ses besoins en redéfinissant certaines de ces méthodes.
- **Extension de programmes déjà conçus à de nouvelles variétés d'objets** : si le module existant structure et manipule des objets d'un type vague **`T`** doté de méthodes virtuelles, il est possible de rajouter des variétés du type **`T`** à celles déjà existantes sous la forme de classes dérivées **`T1`**, **`T2`** ... Par exemple, disposant d'un logiciel qui manipule des objets du type **`Figure`** précédent, il est possible, bien après la conception de ce module, et sans le modifier, de rajouter de nouvelles espèces de figures.

Mélange de plusieurs représentations d'un même type abstrait d'objets

On peut faire cohabiter plusieurs représentations d'un même type abstrait, tout en manipulant les objets de ce type de façon homogène, indépendamment de leur représentation. L'intérêt est ici de pouvoir adapter la représentation à divers cas spéciaux pour améliorer les performances en espace ou en temps. Par exemple, pour des listes, on peut envisager les représentations suivantes :

- La première au moyen d'un tableau et d'une taille effective. Ceci est performant en temps, mais offre des listes de taille limitée.
- La seconde au moyen de maillons créés dynamiquement au fur et à mesure des besoins. Cette solution est plus lente, mais ne limite pas la taille des listes.

Pour permettre la cohabitation de multiples représentations, on peut procéder ainsi : on définit une classe **`T`** ne comportant aucune structure de donnée, qui représente le type abstrait. Les représentations concrètes sont réalisées par autant de classes **`T1`**, **`T2`** ... dérivées de **`T`**. Les méthodes de **`T`** dont la réalisation dépend des structures de données sont virtuelles, et on en rédige la version convenable au sein de chaque classe dérivée.

Souvent, le type **`T`** possède des opérations primitives dont la programmation nécessite la connaissance des structures de données concrètes, et des opérations secondaires que l'on peut entièrement exprimer au moyen des opérations primitives. On peut avantageusement programmer ces opérations secondaires directement au niveau du type abstrait **`T`**, en utilisant les opérations primitives virtuelles, seules ces opérations primitives étant programmées au niveau des réalisations concrètes.

Lorsque le résultat d'une opération est lui-même du type abstrait **`T`**, cela pose quelques problèmes, car il doit être en fait d'un type précis **`T1`** ou **`T2`**, car il n'existe pas d'objet strictement de type **`T`**. L'opération doit produire son résultat par effet de bord, c'est-à-dire en agissant sur un objet, soit l'objet courant, soit un objet passé explicitement en paramètre.

Par exemple, les nombres complexes admettent deux représentations traditionnelles : cartésienne (partie réelle et partie imaginaire, $X+iY$) ou polaire (rayon et angle, $\rho e^{i\theta}$). Pour mêler ces deux représentations en un type général **`Complexe`**, l'opération d'addition devra avoir un des profils suivants :

© IFSIC - Université de Rennes 1 - 2005

```
class Complexe {
...
```

(1) **`void add(Complexe c1)`**
qui réalise **`this = this+c1`**

(2) **`void add(Complexe c1, Complexe c2)`**
qui réalise **`this = c1+c2`**

(3) **`static void add(Complexe c1, Complexe c2, Complexe c3)`**
qui réalise **`c1=c2+c3`**
dans ce cas la méthode est **`static`** car elle n'est pas associée à un objet.

```
}
```


© IFSIC - Université de Rennes 1 - 2003

5 Structures de données génériques

La ***généricité*** est la possibilité de paramétrer la définition de modules logiciels. Un cas fréquent et très utile de généricité consiste à définir des structures de données paramétrées par d'autres types. Par exemple, on peut ainsi définir le type **`Pile`** ***de T***, les piles d'éléments de type quelconque ***T***. Une telle définition s'appelle un ***type générique***. On peut ensuite utiliser ce type générique pour disposer de **`Pile`** ***d'entiers***, de **`Pile`** ***de caractères***, ou de tout autre type passé en paramètre effectif.

Java (à partir de la version 1.5) offre les classes et méthodes génériques.

5.1 Classes génériques

La forme générale d'une classe générique est :

```
class A<T1, T2 ..> { … }
```

Les identificateurs **`T1`**, **`T2`** ... sont les paramètres de généricité : ils peuvent être utilisés comme des identificateurs de classe dans le texte de **`A`**.
Exemple, la pile dont les éléments sont d'un type quelconque :

```
class Pile<T> { // pile générique
  private int  s;
  private Object[] P= new Object[100];
  public Pile() {s=-1;}
  public void empiler(T e) { s++; P[s]=e;}
  public void depiler() { s--;}
  public T sommet() {return (T) P[s];}
  public boolean estVide() {return s==-1;}
}
```

Remarque : curieusement, la version 1.5 de java de permet pas de créer des tableaux de type générique. On est donc obligé d'écrire **`Object[] P= new Object[100]`** au lieu de **`T[] P= new T[100]`**. Ceci oblige à pratiquer un cast **`(T) P[s]`** lors de l'extraction de données de la pile générique.

On peut ensuite déclarer, créer, utiliser de telles piles ayant des éléments d'un type particulier. Par exemple pour une pile d'éléments de type **`Etudiant`** :

```
class TestPile {
  public static void main(String[] z) {
    Etudiant toto = new Etudiant("toto",...);
    Etudiant jules = new Etudiant("jules",...);

    Pile<Etudiant> pEtu = new Pile<Etudiant>();
    pEtu.empiler(toto);
    pEtu.empiler(jules);
    ...
  }
}
```


© IFSIC - Université de Rennes 1 - 2005

5.2 Méthodes génériques

On peut également définir des méthodes génériques. L'exemple suivant est une fonction générique qui étant donné un tableau **`tab`** rend en résultat la chaîne de caractères constituée de la représentation en clair des éléments de **`tab`**, séparés par des blancs. Cette fonction accepte un tableau d'éléments de type quelconque :

```
static <TypeElement> String enClair(TypeElement[] tab){
  if (tab.length==0){return "";}
  String resul=""+tab[0];
  for(int i=1;i<tab.length;i++){resul=resul+" "+tab[i];}
  return resul;
}
```

Les paramètres de généricité, **`TypeElement`** dans l'exemple, sont indiqués juste devant le type du résultat de la fonction.

Un usage fréquent de fonctions génériques sont les ***fonctions statiques de classes génériques***. Le langage Java ne permet pas l'usage des paramètres de généricité dans les fonctions statiques. On contourne cette difficulté en rédigeant une fonction statique générique. Considérons par exemple la classe générique **`Paire`**, représentant une paire dont le premier élément est de type **`T1`** et le second de type **`T2`**. Si on veut doter cette classe d'une fonction statique **`aPartirDe(Tp p, Ts s)`** qui rend en résultat une paire constituée des éléments **`p`** et **`s`**, il faut la rédiger ainsi :

```
class Paire<T1,T2> {
  private final T1 premier;
  private final T2 second;
  private Paire(T1 p, T2 s){premier=p; second=s;}
  public T1 getPremier() {return premier;}
  public T2 getSecond() {return second;}
  public static <Tp,Ts> Paire<Tp,Ts> aPartirDe(Tp p, Ts s){
      return new Paire<Tp,Ts>(p,s);
  }
}
```

5.3 Généricité et types primitifs - *Autoboxing*

En java, les paramètres de généricité sont ***nécessairement des classes***. Ils désignent donc des types d'objets et non des types primitifs (**`boolean`**, **`byte`**, **`char`**, **`short`**, **`int`**, **`long`**, **`float`**, **`double`**). La raison est le choix fait par java de n'autoriser la généricité que pour des données qui sont des ***références***, ce qui permet au compilateur de n'avoir qu'un seul exécutable (**`.class`**) pour une classe générique et non pas une version propre à chaque spécialisation de classe générique.

Pour réaliser des structures de données génériques à partir d'éléments d'un type primitif, il faut encapsuler ce type primitif dans une classe. Par exemple, pour faire une pile d'entiers, il faut utiliser une classe **`Integer`** :

© IFSIC - Université de Rennes 1 - 2003

```
class Integer { // encapsule le type int
  private int v;
  public Integer(int i) {v=i;}
  public int intValue() {return v;}
}
```

et utiliser une **Pile** de **Integer**, car il n'est pas permis d'avoir directement une **Pile** de **int**.

```
Pile<Integer> p = new Pile<Integer>(); // pile d'entiers
```

L'utilisation normale d'une telle pile serait cependant un peu lourde :

```
  p.empiler(new Integer(12));
  p.empiler(new Integer(14));
  p.empiler(new Integer(678));

  while (!p.estVide()) {
    int i = p.sommet().intValue();
    System.out.print(" "+ i);
    p.depiler();
  }
```

Cet exemple montre que pour empiler 12, il faut en réalité empiler un objet de type **Integer** possédant la valeur 12, ce qui alourdit le paramètre effectif en

new Integer(12)

Une lourdeur similaire se produit pour l'obtention de résultats. Le résultat de **sommet** est un objet de type **Integer** et non une valeur de type **int**, ce qui exige l'appel à **intValue**.

Pour alléger l'écriture, java offre la facilité suivante, appelée ***autoboxing*** (enveloppement automatique) :

- La bibliothèque standard définit des classes d'objets correspondant aux types primitifs :
  Integer, **Long**, **Short**, **Byte**, **Float**, **Double**, **Character**, **Boolean**
- Le langage autorise l'usage d'une expression ***e*** d'un type primitif ***t*** partout où son correspondant objet ***T*** est attendu. Cette expression est alors considérée comme équivalente à **new** ***T*****(***e***)**.
  Exemple : on peut écrire directement **p.empiler(12)**, le compilateur traduit cela en **p.empiler(new Integer(12))**.
- Partout où on attend une expression de type primitif ***t***, le langage autorise l'usage d'une expression ***E*** du type d'objet correspondant. Cette expression est alors considérée comme équivalente à ***E*****.*****t*****Value()**.
  Exemple : on peut écrire directement **int i = p.sommet()**, le compilateur traduit cela en **int i = p.sommet().intValue()**.

Cette facilité rend la généricité applicable aux types primitifs sans lourdeur particulière pour l'activité de programmation. Mais il faut avoir conscience qu'elle nuit à l'efficacité d'exécution des programmes Java. Les valeurs de types primitifs provoquent abusivement la création d'objets dans le tas. De plus, l'accès à ces valeurs nécessite une indirection et un appel de méthode.

© IFSIC - Université de Rennes 1 - 2005

Exercice 6

Rédiger une classe générique **`Arbre<InfoNoeud,InfoFeuille>`** qui représente des arbre binaires avec des données de type **`InfoNoeud`** associées aux nœuds et des données de type **`InfoFeuille`** associées aux feuilles.

Principe :
La classe **`Arbre`** possède un attribut de données **`saSorte`** de type énuméré
`enum Sorte {noeud, feuille};`
qui indique si l'arbre est un nœud ou une feuille.
Les nœuds sont réalisés par une classe **`Noeud`** et les feuilles par une classe **`Feuille`**. Pour permettre que les nœuds et les feuilles soient également des arbres, ces classes ***héritent*** de la classe **`Arbre`**.

La classe **`Arbre`** offrira les méthodes suivantes :

- **`estNoeud`** : indique si **`this`** est un nœud,
- **`estFeuille`** : indique si **`this`** est une feuille,
- **`infoNoeud`** : délivre l'information de **`this`** en tant que nœud,
- **`infoFeuille`** : délivre l'information de **`this`** en tant que feuille,
- **`gauche`** : délivre le sous-arbre gauche de **`this`** en tant que nœud,
- **`droite`** : délivre le sous-arbre droite de **`this`** en tant que nœud,
- **`toString`** : délivre une chaîne de caractères représentative de **`this`** sous forme parenthésée.

Rédiger un programme de test qui utilise une spécialisation

`Arbre<String,Integer>`

pour représenter des expressions entières. Les informations associées aux nœuds sont soit **`"+"`** pour signifier l'addition, soit **`"*"`** pour signifier la multiplication. Rédiger une fonction **`eval`** qui évalue une telle expression.

6 Algorithmes génériques

Un ***algorithme générique*** est un algorithme qui peut s'appliquer à des paramètres de types divers dans la mesure où ces types disposent de certaines opérations dont le profil est fixé et qui sont censées satisfaire certaines propriétés. Un exemple simple et classique est le tri d'un tableau : pour pouvoir exprimer le tri, il suffit de disposer d'une fonction qui indique si un élément est ***inférieur*** à un autre.
Java offre la notion d'***interface*** pour exprimer le besoin d'existence de certaines opérations.

© IFSIC - Université de Rennes 1 - 2003

6.1 Interfaces

Une interface est une collection de déclarations de méthodes. On peut la considérer comme un cas limite de classe abstraite : elle ne définit aucun corps de méthode, ni aucun attribut de données. Il n'y figure que des profils de méthodes et éventuellement des déclarations de constantes statiques. Le rôle principal d'une interface est d'établir un "contrat" ou encore un "cahier des charges" que des types d'objets devront satisfaire pour pouvoir participer à certains traitements.

6.1.1 Définition d'interface

Un exemple d'interface est celui de "***parcoureur de collection***". Ce qu'on demande à un tel objet c'est de se comporter comme un "curseur" qui repère un élément au sein d'une collection de données et qui permette de parcourir la collection. Un tel objet peut être abstrait par l'interface **`Parcours`** :

```
interface Parcours<TypeElement> {

  public void tete();
  // effet : positionne this en début de collection
  // ou en fin de parcours si la collection est vide

  public void suivant();
  // prérequis : this n'est pas en fin de parcours
  // effet : positionne this sur l'élément suivant

  public boolean estEnFin();
  // résultat : indique si this est en fin de parcours
  // (au delà du dernier)

  public TypeElement elementCourant();
  // prérequis : this n'est pas en fin de parcours
  // résultat : l'élement désigné par this
}
```

Cet exemple montre la syntaxe d'une interface : le mot **`interface`** remplace le mot **`class`** et il n'y a que des profils de méthodes, terminés par "**`;`**", il n'y a pas de corps. Une interface peut être générique. C'est le cas ici : le type des éléments est indiqué par le paramètre de généricité **`TypeElement`**. Ainsi l'interface **`Parcours`** pourra être mise en œuvre pour des collections d'éléments de type quelconque.

6.1.2 Mise en œuvre d'interface

De même qu'une classe peut hériter d'une autre classe (mot clé **`extends`**), une classe peut ***mettre en œuvre*** une interface. Cela s'indique par le mot-clé **`implements`** : voici par exemple une mise en œuvre de l'interface **`Parcours`**, la classe **`ParcoursDeTableau`** qui permet de parcourir les éléments d'un tableau depuis l'indice 0 jusqu'à l'élément de plus fort indice :

© IFSIC - Université de Rennes 1 - 2005

```
class ParcoursDeTableau<TypeElement>
              implements Parcours<TypeElement> {

  private TypeElement[] T; // tableau objet du parcours
  private int i; // indice courant

  public ParcoursDeTableau(TypeElement[] T) {
  // parcours sur T initialisé à l'indice 0
    this.T=T; i=0;
  }
  public void tete(){
  // effet : positionne this en début de collection
    i=0;
  }

  public void suivant(){
  // prérequis : this n'est pas en fin de parcours
  // effet : positionne this sur l'élément suivant
    i++;
  }

  public boolean estEnFin(){
  // résultat : indique si this est en fin de parcours
  // (au delà du dernier)
    return i==T.length;
  }

  public TypeElement elementCourant(){
  // prérequis : this n'est pas en fin de parcours
  // résultat : l'élement désigné par this
    return T[i];
  }
}
```

On peut également donner une mise en œuvre de **`Parcours`** pour les listes. L'exemple suivant illustre la réalisation de liste au moyen de maillons chaînés. La mise en œuvre des parcours est réalisée par une classe interne **`ParcoursDeListe`**.

```
public class Liste<T> {

  private static class Maillon<TE> {
    Maillon<TE> suivant; TE element;
    Maillon(Maillon<TE> s, TE e) {suivant=s;element=e;}
  }

  private Maillon<T> tete;
  private Maillon<T> queue;

  //=========parcoureur de liste ====================
  public class ParcoursDeliste implements Parcours<T>{
    private Maillon<T> courant;
    public ParcoursDeListe() {courant=tete;}
    public void tete() {courant=tete;}
```


© IFSIC - Université de Rennes 1 - 2003

```
    public void suivant() {
      courant=courant.suivant;
    }
    public boolean estEnFin() {return courant==null;}
    public T elementCourant() {return courant.element;}
  }
  //=================================================

  public Liste() { // liste vide
    tete=null; queue=null
  }

  public Parcours<T> nouveauParcours() {
  // résultat : nouveau parcours initialisé au début
    return new ParcoursDeListe();
  }

  public void ajouteEnQueue(T nouvelElement) {
  // effet : ajoute nouvelElement en queue de this
    if (queue==null) { // cas liste vide
      queue = new Maillon<T>(null,null,nouvelElement);
      tete = queue;
    }
    else { // chaine en queue
      Maillon<T> exDernier = queue;
      queue = new Maillon<T>(null,nouvelElement);
      exDernier.suivant=queue;
    }
  }
}
```

Une interface s'utilise comme une classe abstraite. Étant donné une interface ***I***, on peut déclarer des variables, des résultats ou des paramètres de type ***I***. Tout objet instance d'une classe qui implémente cette interface est compatible avec ces variables, résultats ou paramètres.

En java, une classe ne peut hériter que d'une seule classe : il n'y a pas d'héritage multiple comme dans d'autres langages. En revanche, une classe peut mettre en œuvre un nombre quelconque d'interfaces. Ce choix fait par java répond à la plupart des besoins tout en gardant une grande simplicité au langage. La syntaxe d'une classe qui met en œuvre plusieurs interfaces est :

`class A implements` ***Interface1*****`,`** ***Interface2***`...` **`{`** ... **`}`**

Les algorithmes rédigés en termes d'interfaces sont très généraux, ils peuvent s'appliquer à des situations très diverses, et notamment à des situations qui se présentent bien après la conception de ces algorithmes. Comme exemple simple on peut considérer l'impression d'une collection, sans même savoir ce qu'est précisément cette collection :

© IFSIC - Université de Rennes 1 - 2005

```
static <TE> void afficheCollection(Parcours<TE> p){
  p.tete();
  while(!p.estEnFin()){
    System.out.println(p.elementCourant());
    p.suivant();
  }
}
```

6.2 Utilisation des interfaces pour exprimer des exigences sur les paramètres de généricité

On a parfois besoin de certaines propriétés concernant les paramètres de généricité d'une classe ou d'une méthode générique. Nous prenons ici l'exemple d'un algorithme de tri de tableau. Le type des éléments peut être (presque) quelconque, à condition qu'on puisse décider si un élément est inférieur à un autre.
Pour exprimer une telle contrainte sur le type des éléments, on définit une interface qui déclare une méthode de comparaison :

```
interface Ordonnable<T> {
  public boolean inferieur(T x);
  // résultat : indique si this est inférieur à x
}
```

et l'algorithme de tri exige que le type des éléments du tableau mette en œuvre cette interface. Cela se dit :

```
public static <T extends Ordonnable<T>>
                                    void trier(T[] tab){
// effet : tri tab par ordre croissant selon l'ordre
// défini par la méthode inferieur des éléments de tab
  for (int i=tab.length-1; i>=0; i--) {
    for (int j=1; j<=i; j++) {
      if (tab[j].inferieur(tab[j-1])) {
        T x=tab[j-1]; tab[j-1]=tab[j]; tab[j]=x;
      }
    }
  }
}
```

La forme du paramètre de généricité **`<T extends Ordonnable<T>>`** signifie que **`T`** doit mettre en œuvre l'interface **`Ordonnable<T>`**, c'est-à-dire offrir une méthode **`inferieur`** ayant un paramètre de type **`T`** et un résultat **`boolean`**.

Pour trier un tableau d'éléments de type **`Personne`**, il faut que la classe **`Personne`** soit une mise en œuvre de l'interface **`Ordonnable`**. Nous choisissons ici d'ordonner les personnes par ordre alphabétique de leur nom.

© IFSIC - Université de Rennes 1 - 2003

```
class Personne implements Ordonnable<Personne>{
  public String nom; public int age; public int poids;
  public Personne(String n, int a, int p){
    nom=n; age=a; poids=p;
  }
  public String toString(){
    return "<"+nom+","+age+" ans, "+poids+" kg>";
  }
  public  boolean inferieur(Personne p){
  // résultat : indique si this est avant y par
  // ordre alphabétique de leurs noms
    return nom.compareTo(p.nom)<0;
  }
}
```

Et voici un exemple d'utilisation :

```
public static void main(String[] z){
  Personne[] peuple = {
    new Personne("toto", 25, 80),
    new Personne("tutu", 53, 65),
    new Personne("tata", 15, 47),
    new Personne("jojo", 12, 30)
  };
  System.out.println("peuple = "+enClair(peuple));
  trier(peuple);
  System.out.println("peuple trié selon leur nom = "
                                    +enClair(peuple));
}
```

L'exigence d'une relation d'ordre est un besoin très fréquent, c'est pourquoi une interface **`Comparable`** a été définie dans la bibliothèque standard. Cette interface déclare une fonction **`compareTo`** qui fait simultanément les comparaisons d'infériorité, d'égalité et de supériorité, en rendant un entier respectivement négatif, nul ou positif.

```
interface Comparable<T> {
  public int compareTo(T x);
  // résultat : <0, =0 ou >0 selon que this est
  // respectivement inférieur, égal ou supérieur à x
}
```

Bien évidemment, la sémantique de **`compareTo`** doit respecter les règles d'une relation d'ordre (reflexivité, antisymétrie et transitivité).
Contrairement à ce que nous avons fait dans l'exemple précédent, il vaut mieux, pour plus d'universalité, utiliser cette interface **`Comparable`** plutôt que d'en définir une similaire. Ici il serait préférable de définir ainsi la classe **`Personne`** :

© IFSIC - Université de Rennes 1 - 2005

```
class Personne implements Comparable<Personne>{
  public String nom; public int age; public int poids;
  ...
  public  boolean compareTo(Personne p){
  // résultat : indique si this est avant y par
  // ordre alphabétique de leurs noms
    return nom.compareTo(p.nom)<0;
  }
}
```

Les classes usuelles de la bibliothèque standard, **`Integer`**, **`Double`**, **`String`**... mettent en œuvre l'interface **`Comparable`**.

6.3 Utilisation des interfaces pour transmettre des fonctions en paramètre

Les interfaces permettent de transmettre des fonctions en paramètre et constituent donc un moyen encore plus général d'exprimer des algorithmes génériques. On peut continuer à illustrer cela avec l'exemple du tri. La version précédente ne permet pas de trier un tableau selon plusieurs critères, puisque la fonction de comparaison est liée au type des éléments. Pour trier un tableau selon plusieurs critères, il faut passer en paramètre la fonction de comparaison souhaitée. Une interface permet d'abstraire cette fonction de comparaison :

```
interface Comparaison<T> {
  public boolean inferieur(T x, T y);
 // résultat : indique si x est inférieur à y
}
```

Et dans la nouvelle version du tri, plus générale, un paramètre supplémentaire **`op`** donne accès à la fonction de comparaison à utiliser :

```
public static <T> void trier(T[] tab, Comparaison<T> op) {
// effet : tri tab par ordre croissant selon l'ordre
// défini par la méthode inferieur de op
  for (int i=tab.length-1; i>=0; i--) {
    for (int j=1; j<=i; j++) {
      if (op.inferieur(tab[j],tab[j-1])) {
        T x=tab[j-1]; tab[j-1]=tab[j]; tab[j]=x;
      }
    }
  }
}
```

Avec cette version, pour trier n'importe quel type d'objets selon n'importe quel critère, il suffit de passer en paramètre une instance d'une classe qui met en œuvre l'interface **`Comparaison`**. L'exemple suivant illustre le tri d'un tableau d'éléments de type **`Personne`**, d'abord selon leur âge au moyen de la classe **`CompareAge`**, puis selon leur poids au moyen de la classe **`ComparePoids`** :

© IFSIC - Université de Rennes 1 - 2003

```
class CompareAge implements Comparaison<Personne>{
  public boolean inferieur(Personne x, Personne y){
    return x.age<y.age;
  }
}

class ComparePoids implements Comparaison<Personne>{
  public boolean inferieur(Personne x, Personne y){
    return x.poids<y.poids;
  }
}


public static void main(String[] z){
...
  trier(peuple,new ComparePoids());
  System.out.println("peuple trié selon le poids = "
                                      +enClair(peuple));
  trier(peuple,new CompareAge());
  System.out.println("peuple trié selon l'âge = "
                                     +enClair(peuple));
}
```

Exercice 7

On dispose d'une classe générique **`Table<Tcle,Tval>`** qui réalise des tables de correspondance entre des ***clés de n'importe quel type d'objet*** **`Tcle`** et des ***valeurs de n'importe quel type d'objet*** **`Tval`**. Voici sa spécification :

```
public class Table<Tcle extends Comparable<Tcle>,Tval> {

  public Tval valeurAbsente();
  public static final int maxTaille=5;

  public Table(int c) // table vide de capacité c

  public void associe(Tcle c, Tval v)
  //effet : associe v à c dans this
  // ne fait rien si plus de place

  public void retire(Tcle c)
  // effet : retire l'association de clé c si elle existe
  // ne fait rien si c est absente

  public Tval cherche(Tcle c)
  // résultat : valeur associée à c dans this
  // ou valeurAbsente si la clé est absente
}
```

La ***notion d'égalité*** est nécessaire pour comparer les clés. Pour cela le type **`Tcle`** devra mettre en œuvre l'interface **`Comparable`** de la bibliothèque standard.

© IFSIC - Université de Rennes 1 - 2005

Rédiger un programme de test **`TestTableGenerique`** qui :

Crée une table de correspondance **`notes`** qui associe des noms de personnes de type **`String`** avec des notes d'examen de type **`int`**.

Utilise la table **`notes`**, en tentant d'y ranger les associations :
<**`"dupont"`**,12> <**`"durand"`**,19> <**`"machin"`**,2> <"**`jules"`**,14>
<**`"alfred"`**,19>
puis affiche les valeurs associées aux noms introduits.

Crée une table de correspondance **`damier`** qui associe des positions de damier représentées par des paires d'entiers (i,j) et des contenus de case de damier pouvant prendre pour valeur une chaîne de caractères parmi {**`"pionBlanc"`**,**`"pionNoir"`**,**`"vide"`**}.

Utilise la table, en tentant d'y ranger les associations :
<(1,4),**`"vide"`**> <(2,5),**`"pionBlanc"`**> <(6,3),**`"pionBlanc"`**>
<(4,4),**`"pionNoir"`**> <(2,6),**`"vide"`**>
puis affiche les valeurs associées aux coordonnées introduites.

Exercice 8

La classe **`Matrice`** ci-après offre la somme et le produit de matrices carrées grâce aux méthodes statiques **`somme`**, **`produit`**.

```
class Matrice<T> {

  private Object[][] M; // tableau des éléments

  private Matrice(int t){
  // matrice de taille t, non initialisée
    this.M= new Object[t][t];
  }

  public Matrice(T[][] M){
  // prérequis : M est carré
  // Matrice dotée des éléments de M
    this.M= new Object[M.length][M.length];
    for(int i=0; i<M.length; i++){
      for(int j=0; j<M.length; j++){this.M[i][j]=M[i][j];}
    }
  }

  public int taille(){
  // résultat : la taille de this
      return M.length;
  }

  public static <T> Matrice<T>
      produit(Matrice<T> A, Matrice<T> B, Ope<T> op){
```


© IFSIC - Université de Rennes 1 - 2003

```
// prérequis : A et B matrices carrées de même taille
// résultat : le produit AxB avec les opérations de op
  Matrice<T> C = new Matrice<T>(A.taille());
  for (int i=0; i<C.taille(); i++)
  for (int j=0; j<C.taille(); j++) {
      C.M[i][j] = op.neutre();
      for (int k=0; k<C.taille(); k++) {
        C.M[i][j] = op.add((T) C.M[i][j],
               op.mul((T) A.M[i][k],(T) B.M[k][j]));
      }
  }
  return C;
}

public static <T>  Matrice<T>
   somme ( Matrice<T> A, Matrice<T> B, Ope<T> op) {
// prérequis : A et B matrices carrées de même taille
// résultat : la somme A+B avec les opérations de op
  Matrice<T> C = new Matrice<T>(A.taille());
  for (int i=0; i<C.taille(); i++)
  for (int j=0; j<C.taille(); j++) {
      C.M[i][j] = op.add((T) A.M[i][j],(T) B.M[i][j]);
  }
  return C;
}

public <T> Matrice<T> clone() {
// résultat : une nouvelle matrice copie de this
  Matrice<T> C = new Matrice<T>(taille());
  for (int i=0; i<taille(); i++)
  for (int j=0; j<taille(); j++) {
      C.M[i][j] = M[i][j];}
  }
  return C;
}

public static <T> String toString{
// résultat : this en clair
  String resul="";
  for (int i=0; i<taille(); i++) {
    for (int j=0; j<taille(); j++) {
     resul=resul+" "+M[i][j];
    }
    resul=resul+"\n";
  }
  return resul;
}

public T de(int i, int j) {
// prérequis : 0=<i<this.taille() et 0=<j<this.taille()
```


© IFSIC - Université de Rennes 1 - 2005

```
  // résultat : l'élément (i,j) de this
    return (T) M[i][j];
  }
}
```

Pour généraliser les notions de somme et de produit matriciel, non seulement le type des éléments est un paramètre de généricité de la classe **`Matrice`**, mais encore les opérations à utiliser pour la somme et le produit des éléments sont transmises par le paramètre **`op`** des méthodes **`somme`** et **`produit`**.
Ces opérations sont :

`add` : la somme de deux éléments,
`mul` : le produit de deux éléments,
`neutre` : l'élément neutre de la somme.

Ces opérations sont abstraites au moyen de l'interface **`Ope`** :

```
interface Ope<T> {
  public T neutre();
  public T mul(T x, T y);
  public T add(T x, T y);
}
```

Ainsi **`somme`** et **`produit`** de matrices peuvent être utilisés à diverses fins. Par exemple, on peut traiter des matrices d'entiers de façon classique, les opérations **`add`**, **`mul`** et **`neutre`** étant alors respectivement la somme des entiers, le produit des entiers et l'entier 0.
On peut également utiliser une matrice d'entiers **G** pour représenter un graphe, l'élément **G(i, j)** étant la distance du sommet **i** au sommet **j**, avec **G(i, i)**=0 et **G(i,j)**=**infini** s'il n'y a pas d'arc de **i** vers **j**. Pour calculer les plus courts chemins, les opérations **`add`**, **`mul`** et **`neutre`** doivent dans ce cas être respectivement le minimum de deux entiers, la somme des entiers et la distance "infinie". Avec ces opérations, l'algorithme bien connu de Warshall calcule la matrice des plus courts chemins PCC selon la formule :

PCC = G + G^2 +G^3 + ... + G^{n-1}, n étant le nombre de sommets.

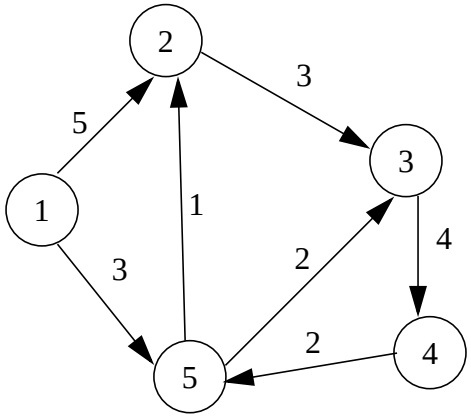
© IFSIC - Université de Rennes 1 - 2003

G =

0	5	∞	∞	3
∞	0	3	∞	∞
∞	∞	0	4	∞
∞	∞	∞	0	2
∞	1	2	∞	0

PCC =

0	4	5	9	3
∞	0	3	7	9
∞	7	0	4	6
∞	3	4	0	2
∞	1	2	6	0



Définir les classes supplémentaires pour pouvoir réaliser la somme et le produit usuel des matrices. Rédiger un morceau de programme qui calcule le produit de deux matrices **`M1`** et **`M2`** supposées déjà construites et initialisées.

Définir les classes supplémentaires pour pouvoir réaliser l'algorithme de Warshall. Rédiger un morceau de programme qui, étant donné un graphe représenté par une matrice **`G`** de taille 5, déjà construite et initialisée, calcule la matrice **`PCC`** des plus courts chemins de ce graphe.

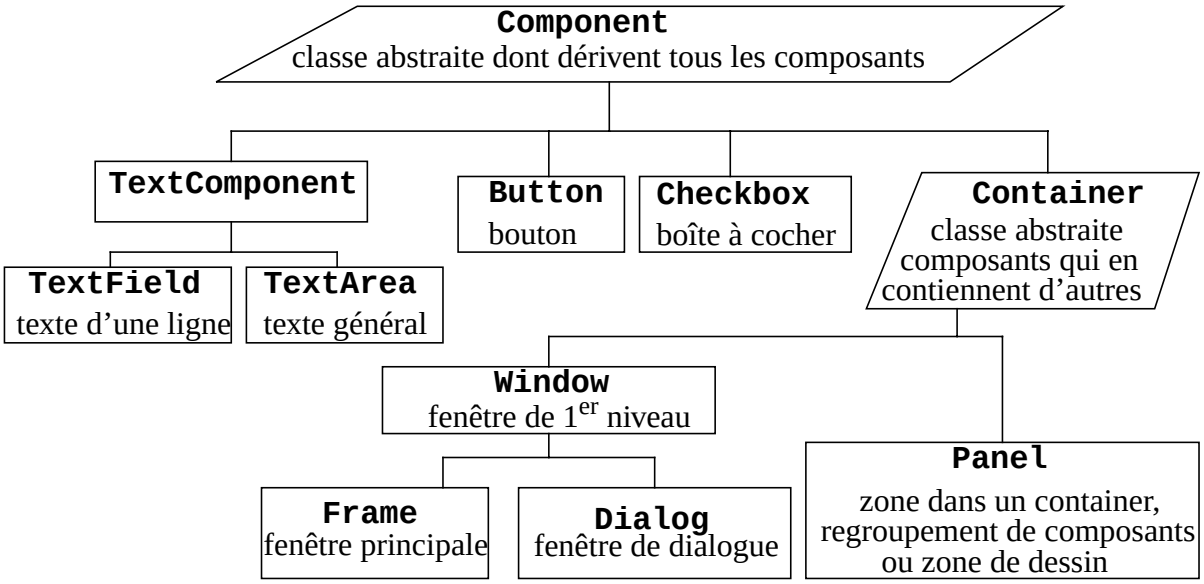
Remarque : $G + G^2 + G^3 + G^4 = (((G)G + G)G + G)G + G$

© IFSIC - Université de Rennes 1 - 2005

7 Interfaces graphiques : paquetages AWT et Swing

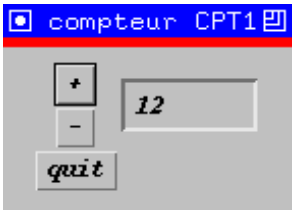
7.1 Organisation générale des interfaces graphiques

Java offre, par ses paquetages AWT (Abstract Window Toolkit) et Swing, le moyen de créer des interfaces à base de fenêtres et de clics sur des boutons. Le schéma suivant montre la hiérarchie des classes AWT qui permettent de construire de telles interfaces.



L'exemple suivant montre l'utilisation de ces classes pour réaliser un compteur commandé par l'interface utilisateur illustrée sur le dessin.

C'est une fenêtre dotée de boutons pour incrémenter un compteur, le décrémenter et quitter l'application. Une zone de texte affiche en permanence l'état du compteur.



1 - Cette fenêtre est réalisée par la fenêtre **`f`** qui est une **`Frame`**.
2 - Les trois boutons sont créés avec en paramètre le texte de leur étiquette.
3 - La zone de texte pour afficher le compteur est créée, avec une taille de 7 caractères.
4 - Pour chaque bouton, on définit une classe destinée à programmer la réaction aux événements générés par le clic sur ce bouton. Ces événements sont du type **`ActionEvent`**, et la méthode qui les traite est définie dans l'interface **`ActionListener`** et s'appelle **`actionPerformed`**. Ces classes implémentent donc cette interface en définissant le corps de la méthode **`actionPerformed`**.
5 - Le constructeur place les composants (boutons et zone de texte) à l'intérieur de la fenêtre, au moyen de la procédure **`Placement`** décrite plus loin.
6 - La méthode **`addActionListener`** de la classe **`Button`** indique quel est le récepteur des événements associé au bouton. Ces récepteurs doivent être des objets de type **`ActionListener`**. Ici ce sont des instances des classes définies en **4**.
7 - La méthode **`pack()`** tasse "au mieux" les composants dans la fenêtre. **`setVisible(true)`** rend la fenêtre visible à l'écran (invisible par défaut).

© IFSIC - Université de Rennes 1 - 2003

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

class FenetreCompteur {
int compteur;

(1) Frame f;
(2) Button boutonIncr=    new Button("+");
(2) Button boutonDecr=    new Button("-");
(2) Button boutonQuit=    new Button("quit");

(3) TextField affichageCompteur = new TextField(7);

class ActionIncr implements ActionListener {
public synchronized void actionPerformed(ActionEvent e)
    {compteur ++; afficherCompteur();}
}

(4) class ActionDecr implements ActionListener {
public synchronized void actionPerformed(ActionEvent e)
    {compteur --; afficherCompteur();}
}

class ActionQuit implements ActionListener {
public synchronized void actionPerformed(ActionEvent e)
    {System.exit(0);}
}

void afficherCompteur() {
  affichageCompteur.setText(String.valueOf(compteur));
}

public FenetreCompteur(String nom) { // constructeur
  f=new Frame("compteur "+nom); compteur=0;
(5)  Placement.p(f,boutonIncr,1,1,1,1);
(5)  Placement.p(f,boutonDecr,1,2,1,1);
(5)  Placement.p(f,boutonQuit,1,3,1,1);
(5)  Placement.p(f,affichageCompteur,2,1,1,2);
(6)  boutonIncr.addActionListener(new ActionIncr());
(6)  boutonDecr.addActionListener(new ActionDecr());
(6)  boutonQuit.addActionListener(new ActionQuit());
(7)  f.pack(); f.setVisible(true);
  afficherCompteur();
}
}

public class TestAWT {
  static public void main(String[] x) {
  new FenetreCompteur("CPT1");
  }
}
```

Remarque : c'est un processus (Thread) spécifique qui s'occupe de donner vie à l'interface utilisateur (afficher les fenêtres, détecter les actions sur la souris et générer les événements). Dans l'exemple, **`main()`**se termine après avoir créé un objet **`FenetreCompteur`**. L'application vit ensuite au titre du processus de gestion de l'environnement, par les appels de méthodes engendrés par les événements.

© IFSIC - Université de Rennes 1 - 2005

7.2 Gestion des événements

Les événements sont classés par thèmes donnant lieu chacun à une classe. Voici quelques unes de ces classes :

	composant générateur et signification
`ActionEvent`	**`Button`** : cliquage, **`TextField`** : touche *Enter*
`MouseEvent`	**`Component`** : mouvements et cliquage de souris
`KeyEvent`	**`Component`** : enfoncement et relâchement de touche
`FocusEvent`	**`Component`** : entrée et sortie du curseur de souris
`TextEvent`	**`TextField`**, **`TextArea`** : modification du texte
`WindowEvent`	**`Window`** : iconification, activation, ouverture, fermeture

etc ...

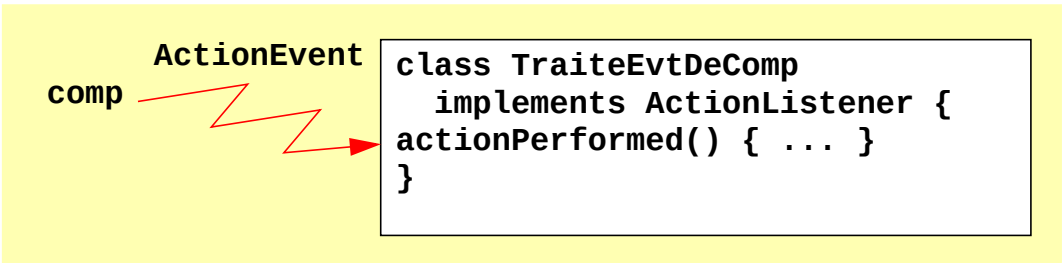
Chaque classe d'événement ***`xxx`*`Event`** est accompagnée d'une interface ***`xxx`*`Listener`** qui définit les méthodes de réaction à ces événements.

```
ActionListener:   actionPerformed(ActionEvent)
MouseListener :   mouseClicked(MouseEvent)
                  mouseDragged(MouseEvent)
                  mouseMoved(MouseEvent)
KeyListener   :   keyPressed(KeyEvent)
                  keyReleased(KeyEvent)
                  keyTyped(KeyEvent)
```

etc ...

Pour récupérer et traiter les événements de type **`xxxEvent`** générés par un composant **`comp`** il faut :

- Définir une classe **`TraiteEvenementsDeComp`** qui implémente **`xxxListener`**. On programme le traitement d'un événement dans le corps d'une méthode de cette classe.
- Inscrire un objet de type **`TraiteEvenementsDeComp`** auprès du composant **`comp`** pour que ce dernier appelle cet objet à l'occasion de chaque événement. Cette inscription se fait au moyen de la méthode **`addxxxListener(xxxListener)`**. Cela se fait souvent dans le constructeur de la fenêtre qui contient le composant **`comp`** :
  `comp.addxxListener(new TraiteEvenementsDeComp`)



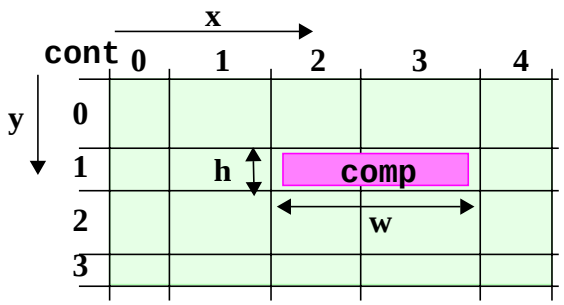
7.3 Placement des composants

Le placement des composants dans un réceptacle (**`Container`**) se fait au moyen de gestionnaires de placement (***layout manager***). Il en existe plusieurs :
`BorderLayout` : place les composants dans cinq zones, le centre et les 4 côtés.

© IFSIC - Université de Rennes 1 - 2003

`CardLayout` : définit des fiches superposées.
`FlowLayout` : range les composants ligne par ligne, de gauche à droite.
`GridLayout` : range les composants dans un tableau à deux dimensions.
`GridBagLayout` : gestionnaire sophistiqué qui range les composants dans une grille topologique en X (horizontalement) et en Y (verticalement). Les emplacements sont désignés par des entiers. Ces entiers ne signifient nullement une mesure de la position mais servent à positionner les composants les uns par rapport aux autres : horizontalement la position ***i+1*** est à droite de la position ***i***, et verticalement ***j+1*** est en dessous de ***j***. Dans cette grille, un composant occupe une certaine zone dont la largeur et la hauteur sont également indiquées en nombre de positions de cellule. Pour chaque composant, on indique comment il est cadré dans sa zone, au centre, à gauche, à droite, en haut ou en bas et quelles sont les marges autour du composant dans sa zone. On indique comment la zone allouée au composant se comporte lors d'une modification de taille de son réceptacle : deux nombres réels fixent le taux d'extension relative de la zone allouée au composant par rapport aux autres composants du même réceptacle.

On fixe également les directions d'extension du composant dans sa zone lorsqu'il dispose de plus de place que nécessaire : aucune extension, extension en largeur, en hauteur ou dans les deux directions.
Ce gestionnaire est très puissant, mais difficile à utiliser.



C'est pourquoi nous avons ici encapsulé son utilisation dans une classe appelée **`Placement`** qui offre les services les plus utiles. L'appel général est :

`Placement.p(`*cont, comp, x, y, w, h, cadrage, t, l, b, r, wx, wy, fill*`)`

cont : réceptacle de type **`Container`** dans lequel est placé le composant
comp : le composant
x, y : position du coin nord-ouest du composant
w, h : largeur et hauteur de la zone allouée au composant
cadrage : cadrage du composant dans sa zone, valeurs possibles :

`GridBagConstraints.CENTER`	au centre
`GridBagConstraints.NORTH`	en haut
`gridBagConstraints.EAST`	à droite
`GridBagConstraints.SOUTH`	en bas
`GridBagConstraints.WEST`	à gauche

t, l, b, r : marge autour du composant dans sa zone, en haut, à gauche, en bas, à droite
wx, wy : poids du taux d'extension horizontale et verticale de la zone allouée
fill : direction(s) d'extension du composant dans sa zone, valeurs possibles :

`GridBagConstraints.NONE`	aucune extension
`GridBagConstraints.HORIZONTAL`	horizontal
`GridBagConstraints.VERTICAL`	vertical
`GridBagConstraints.BOTH`	les deux directions

La classe **`Placement`** offre également des versions simplifiées de la méthode **`p()`**, obtenues en fixant des valeurs par défaut pour certains paramètres :

```
class Placement {
static GridBagLayout placeur= new GridBagLayout();
```


© IFSIC - Université de Rennes 1 - 2005

```
static GridBagConstraints c = new GridBagConstraints();

// procedure generale de placement
//--------------------------------
public static void p( Container cont, Component comp,
     int x, int y, int w, int h, int cadrage,
     int t, int l, int b, int r, double wx, double wy, int fill) {
  cont.setLayout(placeur);
  c.gridx=x; c.gridy=y; c.gridwidth=w; c.gridheight=h;
  c.fill=fill;
  c.anchor=cadrage;
  c.weightx=wx; c.weighty=wy;
  c.insets = new Insets(t,l,b,r);
  placeur.setConstraints(comp, c); cont.add(comp);
};
// placement d'un composant qui ne grossit pas
//---------------------------------------------
public static void p(Container cont, Component comp,
     int x,int y, int w,int h, int cadrage, int t,int l,int b,int r) {
  p(cont, comp, x, y, w, h, cadrage, t, l, b, r,
                       0.0, 0.0, GridBagConstraints.NONE);
};
// placement d'un composant sans marges qui ne grossit pas
//---------------------------------------------------------
public static void p(Container cont, Component comp,
         int x, int y, int w, int h, int cadrage) {
  p(cont, comp, x, y, w, h, cadrage,
              0, 0, 0, 0, 1.0, 1.0, GridBagConstraints.NONE);
};
// placement au centre d'un composant sans marges qui ne grossit pas
//------------------------------------------------------------------
  public static void p(Container cont, Component comp,
         int x, int y, int w, int h) {
  p(cont, comp, x, y, w, h, GridBagConstraints.CENTER,
         0, 0, 0, 0, 1.0, 1.0, GridBagConstraints.NONE);
};}
```

7.4 Quelques méthodes des classes de AWT

```
abstract class Component {
```
`void setVisible(boolean ouiNon);` rend visible ou invisible le composant
`Container getParent();` fenêtre parente du composant
`Dimension preferredSize();` taille préférée du composant
`Dimension getSize();` consulte la taille du composant
`setSize(Dimension d);` change la taille du composant
`Color getBackground();` consulte la couleur du fond
`void setBackground(Color c);` change la couleur du fond
`Color getForeground();` consulte la couleur d'affichage
`void setForeground(Color c);` change la couleur d'affichage

```
...
}
```


© IFSIC - Université de Rennes 1 - 2003

```
abstract class Container extends Component {
  Component add(Component c);
```
ajoute le composant en tant que membre visuel du container
```
  void setLayout(LayoutManager m);
```
associe ce gestionnaire de placement au container
```
...
}
```

```
class Window extends Container{
  Window(Frame parent);
  void pack();
```
arrange au mieux les composants dans la fenêtre
```
  synchronized void dispose();
```
détruit la fenêtre et libère les ressources qu'elle détient
```
...
}
```

```
class Frame extends Window implements MenuContainer {
  Frame(String titre);
  Frame();
  int getCursorType();
```
consulte le type de curseur
```
  void setCursor(int typeCurseur);
```
change le type de curseur
```
  final static int   CROSSHAIR_CURSOR, DEFAULT_CURSOR,
                     E_RESIZE_CURSOR, W_RESIZE_CURSOR, ...
                     HAND_CURSOR, TEXT_CURSOR, WAIT_CURSOR;
```
codage des sortes de curseurs
```
...
}
```

```
class Dialog extends Window {
  Dialog(Frame parent, String titre, boolean modal);
```
si **modal**=**true** : cette fenêtre de dialogue accapare l'attention de l'interpréteur Java tant qu'elle n'est pas détruite par **Dispose()**(peu utile)
```
...
}
```

```
class FileDialog extends Dialog {
  FileDialog(Frame parent, String titre);
  FileDialog(Frame parent, String titre, int load_save);
  final static int LOAD, SAVE;
```
modes du dialogue
```
  String getDirectory();
```
consulte le nom de répertoire
```
  String setDirectory();
```
change le nom de répertoire
```
  String getFile();
```
consulte le nom de fichier
```
  String setFile();
```
change le nom de fichier
```
  String getFilenameFilter();
```
consulte le filtre de présentation des noms de fichiers
```
  String setFilenameFilter();
```
change le filtre de présentation des noms de fichiers
```
...
}
```

```
class Panel extends Container {
  Panel();
...
}
```


© IFSIC - Université de Rennes 1 - 2005

```
class Button extends Component {
  Button(String label);
  Button();
...
}

class Checkbox extends Component {
  Checkbox(String label);
  Checkbox();
boolean getState(); consulte l'état de la boîte à cocher
void setState(boolean etat); change l'état de la boîte à cocher
...
}

class Label extends Component {
  Label(String label);
  Label();
String getText(); consulte le texte affiché
void setText(String t); change le texte affiché
...
}


class TextComponent extends Component {
String getText(); consulte le texte affiché
void setText(String t); change le texte affiché
String getSelectedText(); délivre la zone de texte sélectionnée sur l'écran
int getSelectionStart();
int getSelectionEnd(); délivre l'indice de début / de fin de la zone texte sélectionnée
void setEditable(boolean ouiNon);
        rend la zone de texte éditable ou non depuis le clavier
...
}

class TextField extends TextComponent {
  TextField(int longueur);
  TextField(String texteInitial);
  TextField(String texteInitial, int longueur);
...
}


class TextArea extends TextComponent {
  TextArea(int nbLignes, int nbColonnes);
  TextArea(String texteInitial);
  TextArea(String texteInitial, int nbLignes, int nbColonnes);
...
}


class Color {
  Color(int r, int g, int b); composantes rgb de la couleur
  Color(int code);
  final static Color black, blue, cyan, darkGray, gray,
                green, lightGray, magenta, orange,
                pink, red, white, yellow;
                couleurs usuelles
...
}
```


© IFSIC - Université de Rennes 1 - 2003

7.5 Quelques informations sur la bibliothèque Swing

La bibliothèque Swing est plus récente et plus performante que AWT. Elle ne remplace pas totalement AWT, elle est plutôt “à côté”. Ses composants sont, pour l’essentiel, compatibles avec AWT.
Des composants de fonctionnalité identique à ceux de AWT se retrouvent dans Swing. Ils ont d’ailleurs les mêmes noms préfixés par “**J**” :

```
JWindow JFrame JPanel JFileChooser
JButton JCheckbox JLabel
JTextComponent JTextField JTextArea
JApplet
...
```

Contrairement à une **Frame**, on ne peut pas utiliser une **JFrame** comme conteneur. Les composants doivent être placés dans un conteneur associé, obtenu par la méthode **getContentPane**. Voici par exemple comment programmer le placement d’une zone de texte dans une **JFrame** :

```
...
JFrame f = ...;
JTextField texte = new JTextField(20);
...
Placement.p(f.getContentPane(), texte, 1,1,1,1);
...
```

Pour donner un aperçu de la grande variété des composants offerts par Swing, voici quelques classes de cette bibliothèque :

JScrollPane : rectangle d’observation (*view port*) qui encapsule un composant et que l’on peut déplacer au moyen de curseurs (*scroll bar*).

JMenuBar, **JMenu**, **JMenuItem** : pour disposer des menus déroulants. On peut placer une barre de menus (**JMenuBar**) à une fenêtre de type **JFrame** ou **JApplet**. On peut ajouter des menus (**JMenu**) à la barre de menus, composés de rubrique (**JMenuItem**). Chaque rubrique peut être dotée d’écouteurs de type **ActionListener**.

JTree : visualisation de données sous forme d’un graphe arborescent.

JSlider : barrette dotée d’un curseur pour acquérir des données numériques de type “contrôle de niveau”.

JTabbedPane : affichage à onglets.

JTable : visualisation de tables.

...

© IFSIC - Université de Rennes 1 - 2005

Exercice 9

On considère l'exemple du compteur :

Modifier le programme pour permettre à l'utilisateur de définir un incrément **`delta`** parmi 1, 2, 3 ou 4, au moyen de 4 boutons. Les boutons **+** et **-** ayant pour effet d'ajouter ou de retrancher **`delta`**.

Méthode imposée :
Chaque bouton **`delta1`**, **`delta2`**, **`delta3`** et **`delta4`**, doit provoquer l'affectation d'une valeur d'incrément respectivement égale à 1, 2, 3 et 4. Créer un modèle unique (une classe) **`ActionDelta`** d'écouteur de bouton delta et instancier ce modèle en un écouteur spécifique à chaque bouton en passant la valeur d'incrément en paramètre de construction.

© IFSIC - Université de Rennes 1 - 2003

Exercice 10

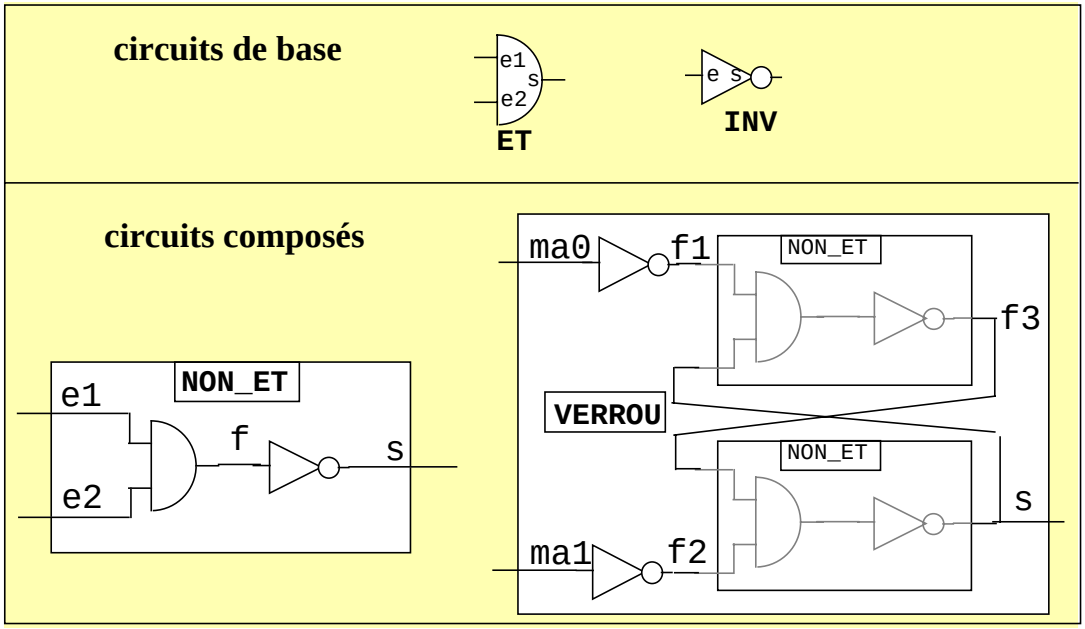
Simulateur de circuits logiques

On désire programmer un simulateur de circuits logiques (simple), en utilisant au mieux les notions de Java.

Les circuits logiques sont soit des ***circuits de base*** (par exemple inverseur **`INV`**, porte et **`ET`**) soit des ***circuits composés***, résultant d'un assemblage d'autres circuits.

Les composants d'un circuit composé sont reliés par des ***fils*** : par exemple, le circuit composé **`NON_ET`** utilise le fil **`f`** pour connecter la sortie d'une instance de **`ET`** à l'entrée d'une instance de **`INV`**.

Les ***broches***, fils d'entrée et de sortie d'un modèle de circuit, permettent de connecter une instance de circuit à des fils effectifs qui le relient à d'autres circuits.



Un fil est porteur d'une valeur logique 0, 1 ou X, X représentant la valeur indéterminée.
Dans le simulateur, ces valeurs seront représentées par les caractères **`'0'`**, **`'1'`** et **`'X'`**.
Initialement, les fils portent une valeur indéterminée (**`'X'`**).

© IFSIC - Université de Rennes 1 - 2005

Principe de fonctionnement du simulateur

Le but du simulateur est de faire fonctionner les circuits par instants successifs (unité de temps simulé), ou ***étapes***. Chaque étape comporte deux phases :

Première phase
Chaque circuit de base possède une méthode d'évaluation, appelée **`topEval()`**, qui définit son comportement. Le simulateur appelle les méthodes **`topEval()`** de tous les circuits de base, ce qui a pour effet d'attribuer une ***valeur future*** aux fils de sortie en fonction des ***valeurs courantes*** portées par les fils d'entrée (ceci ne modifie pas la valeur courante du fil).

Deuxième phase
Chaque fil possède une méthode appelée **`actualise()`** qui met à jour sa valeur courante au moyen de sa valeur future. Le simulateur appelle, pour chaque fil, sa méthode **`actualise()`**.

Une analyse du problème a conduit aux remarques suivantes, qui doivent servir de guide à la rédaction du logiciel :
Les circuits et les fils ont des caractéristiques qui en font tout naturellement des objets.
Les circuits composés sont composés de circuits et de fils.
Les broches d'un circuit désignent les fils qui relient ce circuit à d'autres circuits. Cette désignation peut avantageusement être installée lors de la création du circuit, au moyen des fils effectifs d'interconnexion passés en paramètre du constructeur.

Le simulateur doit avoir accès à la liste de tous les circuits de base et à la liste de tous les fils. Il est judicieux de faire de ces listes des membres statiques dans lesquelles chaque objet, circuit de base ou fil, se rajoute lors de sa création.

1ère partie

Rédiger les classes nécessaires pour disposer des circuits suivants :
circuits de base **`INV`** et **`ET`**,
circuits composés **`NON_ET`** et **`VERROU`**.

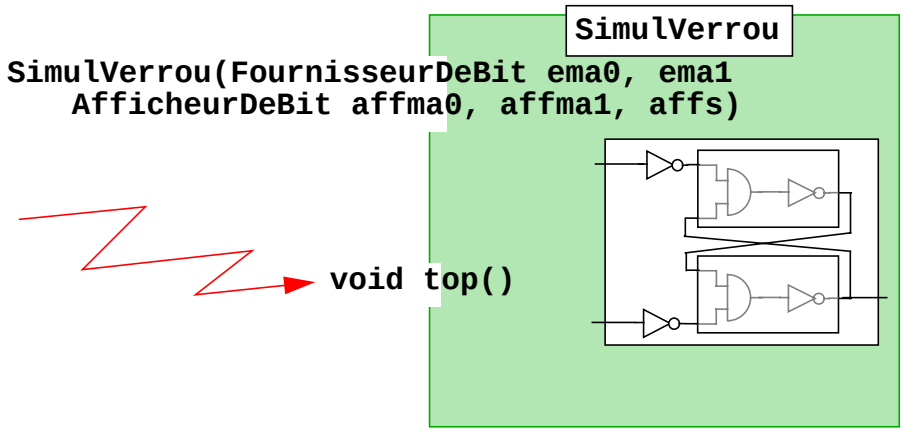
2e partie

Rédiger le simulateur sous forme d'une classe **`SimulVerrou`**. Cette classe offre une méthode **`top()`** qui assure une étape de simulation. La simulation complète sera faite par une séquence d'appels à la méthode **`top()`**.
La classe **`SimulVerrou`** constitue le "cœur" de l'application. Il est essentiel qu'elle soit ***indépendante de toute interface utilisateur particulière***. Pour cela elle disposera de deux "fournisseurs de bits" pour lire les séquences de valeurs à placer sur les fils d'entrée **`ma0`** et **`ma1`** et de trois "afficheurs de bits" pour afficher les valeurs des entrées **`ma0`**, **`ma1`** et de la sortie **`s`**.
Ces fournisseurs de bits et ces afficheurs de bits seront définies comme **`interface`** (très simples), passés en paramètre du constructeur du simulateur. Un usage particulier (voir 3e partie) consistera à implémenter ces interfaces.

© IFSIC - Université de Rennes 1 - 2003

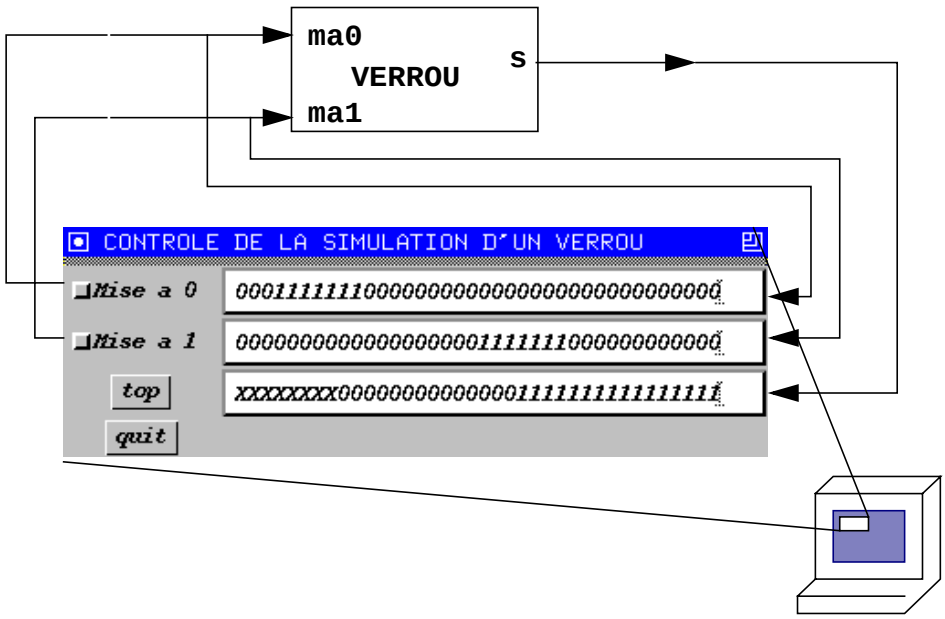
```
interface AfficheurDeBit{
   void ecrire(char b)
}
```

```
interface FournisseurDeBit{
   char lire()
}
```



3e partie

Réaliser l'interface utilisateur illustrée sur la figure. Deux boîtes à cocher (**CheckBox**) servent à indiquer la valeur présente de **ma0** et **ma1** (relâché signifie **0** et enfoncé signifie **1**. Rien n'est prévu pour injecter la valeur indéterminée **X**). Un bouton ***top*** sert à provoquer une étape de simulation. Trois zones de texte servent à afficher la suite des valeurs de **ma0**, **ma1** et **s**. Un bouton ***quit*** permet de terminer l'application.



Cette fenêtre de contrôle offre les classes internes qui ***implémentent*** les interfaces **AfficheurDeBit** (écriture d'un caratère dans une des zones de texte) et **FournisseursDeBit** (consultation des boîtes à cocher). Elle crée une instance de l'application **SimulVerrou** et répercute les clics de souris en appel de la méthode **top()**.

© IFSIC - Université de Rennes 1 - 2005

8 Parallélisme en Java : `Thread`

Pour exprimer des traitements parallèles, Java offre le moyen de créer des ***processus***. Ces processus communiquent en partageant des objets et se synchronisent au moyen de ***moniteurs*** qui sont une variante des moniteurs de Hoare.

8.1 Création de processus

Java offre deux façons de créer des processus : par création d'un objet qui hérite de la classe **`Thread`**, ou par exécution de la primitive **`new Thread()`** sur un objet qui implémente l'interface **`Runnable`**. Nous n'utiliserons ici que la première forme, car la seconde n'ajoute rien d'essentiel. La classe **`Thread`** déclare une méthode virtuelle **`run()`** dans laquelle on rédige le programme principal du processus. L'exemple suivant crée deux processus, un sur le modèle **`A`** qui imprime **`A0`** ... **`A7`**, et un autre sur le modèle **`B`** qui imprime **`B0`** ... **`B7`**. La création est faite par **`new A()`** et **`new B()`**. Cependant cette création ne lance pas par elle-même l'exécution. Il faut de plus appeler la méthode **`start()`** de la classe **`Thread`** qui lance effectivement le processus sur l'exécution de la méthode **`run()`**.

```
class A extends Thread {
public void run() {
  for(int i=0; i<8; i++) { System.out.print("A"+i+" ");
    try {sleep(100);} catch (InterruptedException e) {};
  }
}
}
class B extends Thread {
public void run() {
  for(int i=0; i<8; i++) { System.out.print("B"+i+" ");
    try {sleep(200);} catch (InterruptedException e) {};
  }
}
}

public class TestThread1 {
public static void main(String[] x) {
    new A().start(); new B().start();
}
}
```

Les processus sont exécutés "en parallèle". Dans le cas (usuel) d'une machine à nombre de processeurs limité (généralement limité à 1), un mécanisme de partage des processeurs fait progresser tour à tour les processus. Pour permettre d'intervenir sur l'allocation du processeur, Java offre un système de priorité que nous n'utiliserons pas ici.
Dans l'exemple, on a réglé la "vitesse" d'exécution des processus au moyen de la méthode **`sleep(int t)`** qui met en attente un processus pendant **`t`** milli-secondes. Il en résulte l'entrelacement suivant des impressions de **`A`** et de **`B`** :

```
A0 B0 A1 B1 A2 A3 B2 A4 A5 B3 A6 A7 B4 B5 B6 B7
```


© IFSIC - Université de Rennes 1 - 2003

Dans l'exemple précédent, chaque processus est créé avec un modèle qui lui est propre, la classe **A** et la classe **B**. On peut créer plusieurs processus sur un même modèle, avec éventuellement des paramètres de création, comme le montre la version suivante, totalement équivalente à l'exemple précédent :

```
class Impr extends Thread {
String txt; int periode;
public Impr(String t, int p){txt=t; periode=p;}
public void run() {
  for(int i=0; i<8; i++) { System.out.print(txt+i+" ");
    try {sleep(periode);}catch(InterruptedException e){};
  };
}
}

public class TestThread2 {
public static void main(String[] x) {
new Impr("A",100).start(); new Impr("B",200).start();
} }
```

8.2 Exclusion et moniteurs : `synchronized`

8.2.1 Exclusion

Pour que plusieurs processus puissent se partager sainement des objets afin de coopérer ou d'utiliser des ressources, il faut pouvoir limiter le parallélisme en assurant qu'un certain objet ne subisse pas en même temps plusieurs séquences d'actions : c'est ce qu'on appelle assurer l'exclusion mutuelle de ces actions quant à cet objet. L'exemple suivant illustre un besoin d'exclusion. Deux processus impriment l'un des "**BONJOUR**" et l'autre des "**AU REVOIR**".

```
public class TestThread3 {
  public static void main(String[] x) {
    new Impr("BONJOUR ").start();
    new Impr("AU REVOIR ").start();
  }
}

class Impr extends Thread {
String txt;
public Impr(String t){txt=t;}
public void run() {
  for(int j=0; j<2; j++) {
    for(int i=0; i<txt.length(); i++) {
      try {sleep(100);} catch (InterruptedException e) {};
       System.out.print(txt.charAt(i));
    }
  }
}}
```


© IFSIC - Université de Rennes 1 - 2005

Cette programmation n'assure pas l'exclusion de l'impression pour toute la durée de l'impression d'une chaîne significative. On obtient le résultat suivant dans lequel apparaît un mélange incontrôlé des caractères des deux textes :

```
BAOUN JROEUVRO IBRO NAJUO URRE VOIR
```

Pour assurer les exclusions, Java offre la primitive **`synchronized`**. En Java ***tout objet est susceptible d'être un motif d'exclusion***. La syntaxe générale d'une exclusion relative à un objet **`obj`** est :

`synchronized(obj) {` *bloc d'instructions* **`}`**

Cette construction assure que ce bloc d'instructions n'est pas exécuté avant que toute exécution en cours sous la coupe d'un **`synchronized(obj)`** ne soit achevée.

On peut alors programmer comme suit l'exclusion souhaitée :

```
class Exclusion {};

class PusImpr2 extends Thread {
String txt;
static Exclusion exclusionImpression = new Exclusion();

public PusImpr2(String t){txt=t;}

public void run() {
  for(int j=0; j<2; j++) {
  synchronized(exclusionImpression){
    for(int i=0; i<txt.length(); i++) {
      try {sleep(100);} catch(InterruptedException e) {};
      System.out.print(txt.charAt(i));
    }
    System.out.print(" ");
  }
  }
}
}
```

On a défini une classe **`Exclusion`**, dont le seul rôle est de permettre de créer de "purs objets d'exclusion". Un objet **`exclusionImpression`** de type **`Exclusion`** est utilisé comme argument de **`synchronized`** afin d'assurer l'exclusion sur l'impression pour la durée de la boucle d'impression. Cette version imprime un résultat plus convenable :

```
BONJOUR AU REVOIR BONJOUR AU REVOIR
```

Remarque : la classe **`Exclusion`** peut sembler artificielle. Elle est vide, sans aucun membre de donnée ni aucune méthode. Elle n'est pas si artificielle que cela. En fait elle modélise l'essence même de ce qu'est un objet : sa principale propriété est d'être "égal à lui-même et différent des autres" et c'est effectivement la seule propriété d'un motif d'exclusion.

© IFSIC - Université de Rennes 1 - 2003

8.2.2 Moniteurs

L'exclusion peut porter sur une méthode complète. La syntaxe est dans ce cas :

```
class A {
...
synchronized … P(…) { ... }
...
}
```

Le motif d'exclusion est alors ***l'objet courant de classe*** **A**, et l'exclusion est garantie pour toute l'exécution de la méthode. La classe **A** est ainsi un modèle de ***moniteur*** (au sens de Hoare) et les méthodes dotées de l'attribut **synchronized** sont les ***entrées*** d'un tel moniteur. Cette forme est donc strictement équivalente à :
… **P(…) { synchronized(this) { ... } }**

Quand c'est possible, et c'est pratiquement toujours le cas, il est préférable d'utiliser cette forme, qui force à structurer l'application de telle manière que les problèmes de synchronisation relatifs à un même thème soient centralisés au sein d'un moniteur, plutôt que dispersés dans plusieurs morceaux de textes. Le motif d'exclusion est alors un moniteur dont les membres de données constituent tout ou partie de l'état de la chose partagée et dont les méthodes sont les actions à exécuter en exclusion. Pour l'exemple précédent, cela donne :

```
class MoniteurImpression {
synchronized void imprTexte(String txt) {
  for(int i=0; i<txt.length(); i++) {
    try {Thread.sleep(100);}
    catch(InterruptedException e) {};
    System.out.print(txt.charAt(i));
  };
}
}


class Impr extends Thread {
String txt;
static MoniteurImpression m1 = new MoniteurImpression();

public Impr(String t){txt=t;}

public void run() {
  for(int j=0; j<2; j++) { m1.imprTexte(txt);};
}
}
```


© IFSIC - Université de Rennes 1 - 2005

8.3 Attente explicite : `wait` - `notify`

8.3.1 Attente qu'une condition soit satisfaite : `wait()`

À l'intérieur d'un moniteur (c'est à dire d'une méthode ou d'un bloc qualifié par **`synchronized`**), un processus peut se mettre en attente au moyen de **`wait()`**. Cette primitive est susceptible de déclencher, pendant l'attente, une exception de type **`InterruptedException`**, ce qui oblige souvent à utiliser la forme :

```
try {wait();} catch(InterruptedException e) {...};
```

Lorsqu'un processus exécute **`wait()`** sur un objet **`obj`**, cela relâche l'exclusion sur l'objet **`obj`**, de sorte que d'autres processus puissent acquérir cette exclusion et venir le réveiller.

8.3.2 Réveil des processus : `notify()` et `notifyAll()`

Un processus sort de l'attente lorsqu'un autre processus exécute **`notify()`** au sein de ce même moniteur. Cette primitive existe sous deux formes :

`notify()` : relance ***un*** processus en attente dans ce moniteur,
`notifyAll()`: relance ***tous les*** processus en attente dans ce moniteur.

À chaque moniteur est associée une file d'attente de processus en attente. Il semblerait que les processus soient sortis de la file selon leur ordre d'arrivée, mais cela n'est pas clairement spécifié dans les documents actuels sur le langage : il est donc préférable de ne pas en tenir compte dans la programmation. On peut exécuter **`notify()`** même si aucun processus n'est en attente : cela ne fait rien dans ce cas.

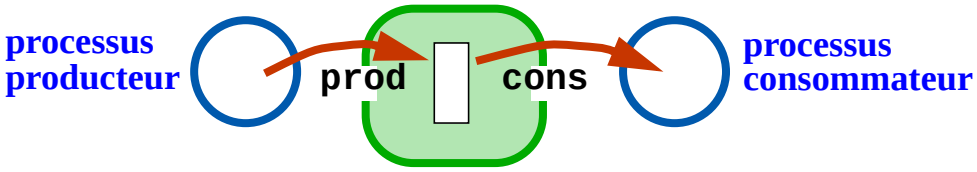
L'exemple suivant illustre la programmation du classique tampon producteur-consommateur à une place.

```
class MoniteurProdCons {
String tampon; boolean estVide=true;

synchronized void prod(String m) {
  if(!estVide){ System.out.println("PRODUCTEUR ATTEND");
     try {wait();} catch(InterruptedException e) {};
  };
  System.out.println("PRODUIT   : " + m);
  tampon=m; estVide=false; notify();
}

synchronized String cons() {
  if(estVide){ System.out.println("CONSOMMATEUR ATTEND");
     try {wait();} catch(InterruptedException e) {};
  };
  System.out.println("CONSOMME  : " + tampon);
  String resul=tampon; estVide=true; notify();
  return resul;
}
}
```


© IFSIC - Université de Rennes 1 - 2003

Le modèle de tampon est mis en œuvre par la classe **MoniteurProdCons** qui offre deux entrées, **prod()** pour produire et **cons()** pour consommer. Un processus qui veut produire est mis en attente si le tampon est plein, un processus qui veut consommer est mis en attente si le tampon est vide.

Une utilisation possible est illustrée ci-dessous. La classe **Producteur** est un modèle de processus qui produit trois messages, et la classe **Consommateur** est un modèle de processus qui consomme trois messages. On a ajusté le débit de la production au moyen de **sleep()** de façon à avoir un comportement non trivial dans lequel apparaissent des attentes aussi bien du producteur que du consommateur.




```
class Producteur extends Thread {
MoniteurProdCons tampon;
public  Producteur (MoniteurProdCons t) { tampon=t;}
public void run() {
     tampon.prod("message1"); tampon.prod("message2");
     try {sleep(100);} catch(InterruptedException e) {};
     tampon.prod("message3");
}}

class Consommateur extends Thread {
MoniteurProdCons tampon;
public  Consommateur(MoniteurProdCons t) { tampon=t;}
public void run() {
   tampon.cons(); tampon.cons(); tampon.cons();
}}
```

Le programme principal crée un tampon et une paire de processus producteur/consommateur qui communiquent par ce tampon :

```
public class TestProducteurConsommateur {
static public void main(String[] x) {
    MoniteurProdCons tampon = new  MoniteurProdCons();
    new Producteur(tampon).start();
    new Consommateur(tampon).start();
}}
```

L'exécution de cet exemple donne la trace suivante :

```
PRODUIT   : message1
PRODUCTEUR ATTEND
CONSOMME  : message1
PRODUIT   : message2
CONSOMME  : message2
CONSOMMATEUR ATTEND
PRODUIT   : message3
CONSOMME  : message3
```

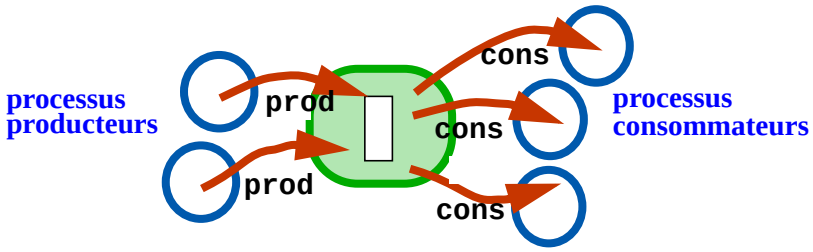

© IFSIC - Université de Rennes 1 - 2005

Choix entre `notify()` ou `notifyAll()`

L'usage de **`notifyAll()`** est plus général que l'usage de **`notify()`**. En effet, dans le cas général, les processus attendent pour diverses raisons et doivent être réveillés chaque fois que l'état change et a quelque chance de satisfaire l'un d'entre eux. Les processus réveillés "à tort" n'ont qu'à se remettre en attente par une programmation de la forme :

```
while (etat non satisfaisant) {
  try {wait();} catch(InterruptedException e) {};
}
```

L'usage de **`notify()`** n'est correct que dans certains cas particuliers. Ainsi, la programmation précédente du tampon producteur consommateur ne fonctionne que s'il n'existe qu'un seul producteur et un seul consommateur pour un tampon. Si on considère un tampon acceptant plusieurs producteurs et/ou plusieurs consommateurs, il risque de se produire le mauvais fonctionnement suivant : deux producteurs p1 et p2 sont attente, un consommateur vient consommer la donnée du tampon, réveille le producteur p1 qui produit sa donnée et malencontreusement réveille p2 qui écrase la donnée de p1.
La programmation correcte d'un tampon pour plusieurs producteurs et consommateurs doit pratiquer **`notifyAll()`** et l'attente dans une boucle :




```
class MoniteurProdCons {
  String tampon; boolean estVide=true;

  synchronized void prod(String m) {
    while(!estVide){
      try {wait();} catch(InterruptedException e) {};
    }
    tampon=m; estVide=false; notifyAll();
  }

  synchronized String cons() {
    while(estVide){
      try {wait();} catch(InterruptedException e) {};
    }
    String resul=tampon; estVide=true; notifyAll();
    return resul;
  }
}
```


© IFSIC - Université de Rennes 1 - 2003

8.3.3 Utilisation - Analogie avec les Moniteurs de Hoare

Ce mécanisme constitue un cas particulier des ***conditions*** offertes par les moniteurs de Hoare. Un moniteur de Hoare permet de déclarer plusieurs conditions $\mathbf{c_1}$, $\mathbf{c_2}$ ... sur lesquelles les processus peuvent attendre par **attendre(c_i)**, et être relancés par **reprendre(c_i)**.
Du point de vue méthode de programmation, les conditions s'utilisent ainsi : si une certaine condition logique, exprimée par un prédicat **P** concernant l'état doit être satisfaite pour qu'une action puisse être poursuivie, on associe une condition $\mathbf{c_p}$ à ce prédicat. Aux endroits du programme où **P** doit être satisfait, on écrit quelque chose comme :

... si non P alors attendre(c_p) fsi; **/* P** est vrai ***/ ...**

et aux endroits où on sait que **P** devient vrai, on écrit :

... /* P est vrai ici ***/ reprendre(c_p) ...**

Si ceci est partout respecté, c'est-à-dire si **P** est vérifié devant tout **reprendre(c_p)**, l'assertion **P** est bien évidemment vérifiée derrière tout **attendre(c_p)** sous réserve que le langage assure le passage du contrôle (séquentiel, car on est au sein de l'exclusion du moniteur) aussitôt au processus réveillé. D'un point de vue de la preuve formelle, la condition $\mathbf{c_p}$ agit comme un "tuyau" qui transmet la vérité de l'assertion **P** entre divers points du texte de programme.

La différence essentielle avec les moniteurs de Java est que ces derniers ont ***une seule condition*** par moniteur, non déclarée car définie implicitement. Avec une seule condition, on se ramène au cas général en re-testant après chaque attente que le prédicat **P** est vrai et en se remettant en attente sinon. La programmation devient quelque chose comme :

```
... while (!P) {wait();} /* P est vrai */ ...
```

et aux endroits où une assertion attendue devient vraie :

```
... /* P ou Q ou R ou ... */ notifyAll();
```
(parfois `notify()`)

Le choix entre `notify()` et `notifyAll()` n'est pas toujours évident. Si on respecte strictement la méthode suggérée ici, on peut toujours utiliser `notifyAll()` : cela risque simplement de conduire à de mauvaises performances à cause de réveils inutiles de processus.

Allocation du moniteur après `notify()` :

Après un `notify()`, plusieurs processus sont concurrents pour utiliser le moniteur. Cependant le langage doit assurer le maintien de l'exclusion sur ce moniteur (c'est obligatoire si on veut pouvoir affirmer quoi que ce soit sur l'état du moniteur). Avec les moniteurs de Hoare, la priorité est officiellement attribuée au processus réveillé : cela peut sembler être une sur-spécification dont on se passerait volontiers, mais c'est nécessaire pour assurer la vérité de l'assertion associée (mentalement par le programmeur) à la condition derrière tout **attendre()**, car si le processus réveilleur est poursuivi après le réveil, son action peut rendre caduque cette assertion. Avec les moniteurs de Java, la règle n'a pas besoin d'être aussi précise car il n'y a qu'une condition qui n'est pas en général associée à une assertion précise et les assertions sont destinées à être re-testées par les processus réveillés. Expérimentalement, la priorité

© IFSIC - Université de Rennes 1 - 2005

semble être donnée au réveilleur, qui continue donc après le **`notify()`**. Un processus réveillé n'est repris que lorsque le réveilleur quitte l'exclusion, c'est-à-dire sort du moniteur ou bien se met en attente par **`wait()`** dans ce moniteur. De toutes façons, il est toujours préférable de programmer avec le moins d'hypothèses possibles sur la sémantique du langage quant à ces détails.

8.4 Réflexions sur l'usage des processus

8.4.1 Nature des processus

Un ***processus n'est pas un objet***, ou du moins il n'est pas un objet comme les autres. C'est une exécution, une activité, donc quelque chose de plus difficile à percevoir qu'un objet. Certes il y a bien un objet associé au processus, l'objet de classe **`Thread`** qui lui donne naissance. Mais cet objet n'est que le point de départ du processus, son programme principal. Le propre d'un processus c'est de se promener d'objet en objet. Une image assez juste est la suivante : les objets sont des fleurs et les processus sont des abeilles qui butinent ces fleurs.

Cela se perçoit concrètement dans le langage : il y a la notion de ***processus courant***, accessible par la méthode **`currentThread()`**. C'est une méthode statique de la classe **`Thread`**, qui rend en résultat l'objet de classe **`Thread`** associé au processus qui l'exécute. Cette méthode s'utilise donc généralement sous la forme :

```
... Thread.currentThread() ...
```

Ainsi il y deux notions de choses "courantes" bien distinctes qu'il ne faut pas confondre :

`this` : l'objet courant, sur lequel a lieu l'exécution (la fleur)

`Thread.currentThread()` : le processus qui exécute (l'abeille).

Certaines primitives concernant les processus sont des méthodes de la classe **`Thread`**, par exemple :

`static void sleep(int t) throws InterruptedException`
attend **`t`** millisecondes

`void join() throws InterruptedException`
attend qu'un processus termine

`int getPriority()`
priorité courante du processus pour l'usage du processeur

© IFSIC - Université de Rennes 1 - 2003

`void setPriority(int p) throws IllegalArgumentException`
change la priorité du processus
`void stop()` termine un processus
...
Pour utiliser ces méthodes, il faut employer une des formes :
`Thread.sleep(100)`
`Thread.currentThread().getPriority()`

D'autres primitives concernant les processus sont des méthodes de la classe **`Object`**, ce qui signifie qu'elles s'appliquent depuis n'importe quelle classe, car toutes dérivent de la classe **`Object`**. Ce sont notamment :
`wait()`, **`notify()`**, **`notifyAll()`**.
Ces méthodes utilisent le verrou d'exclusion et la file d'attente de condition associés à tout objet. Si on n'indique rien de plus, elles utilisent donc ces éléments associés à l'objet courant, **`this`**. Cependant, le langage étant totalement orthogonal, rien n'empêche de les utiliser sur d'autres objets :

```
synchronized (unObjet) {
      ... unObjet.wait(); ... unObjet.notify(); ...
}
```

Pour des raisons de structuration des programmes, il vaut mieux si possible éviter cette forme.

8.4.2 Intérêts des processus

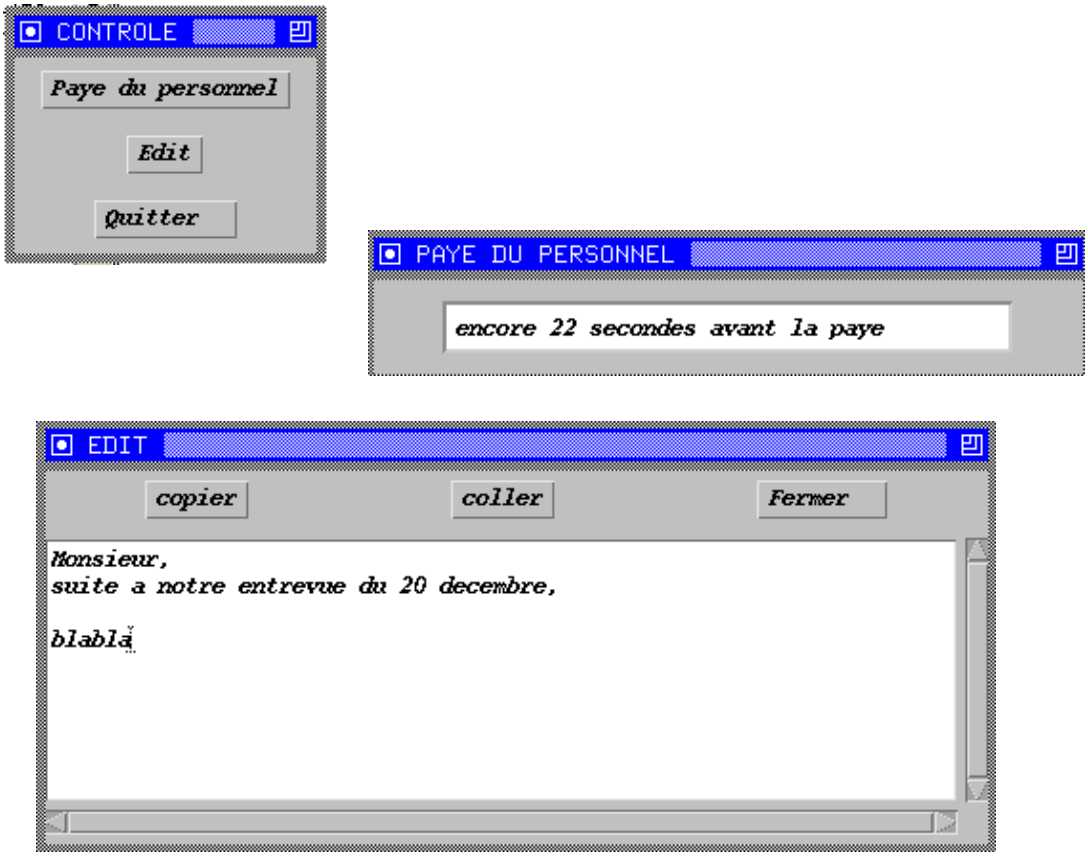
L'usage des processus offre divers avantages, souvent mélangés, de sorte qu'il est difficile de dire si on les utilise pour telle raison ou pour telle autre. On peut distinguer trois grandes catégories d'usages :

- ***Interactivité*** : l'extérieur étant par nature parallèle, il est souhaitable que l'intérieur le soit également. Par exemple, après avoir lancé une copie de disquette ou le chargement d'une image d'avion depuis le Web, il est agréable de ne pas être bloqué et pouvoir faire de l'édition de texte.
- ***Structure de contrôle plus riche*** : chaque processus possède son propre contexte de travail, son état de contrôle qui indique où il en est dans le programme, c'est-à-dire un "compteur ordinal" au sens large. Avec plusieurs processus on dispose donc de plusieurs compteurs ordinaux, et donc d'une plus grande richesse d'expression. Dans ce cas ce n'est pas le parallélisme qui est exploité : ces processus ne servent qu'à "attendre" et n'ont jamais l'occasion de tourner en même temps. En revanche, ils peuvent attendre tout en conservant leur état de contrôle et cela permet une meilleure modularité des textes des programmes.
- ***Calcul parallèle*** : les processus permettent d'exprimer des algorithmes parallèles de calcul, et on peut espérer que les performances soient meilleures si le parallélisme est effectif sur plusieurs processeurs.

Interactivité :

L'exemple suivant illustre l'usage des processus pour raison d'interactivité. Il offre la possibilité de lancer, depuis une fenêtre de contrôle, la "paye du personnel" tout en permettant à l'opérateur de faire de l'édition de texte.

© IFSIC - Université de Rennes 1 - 2005



La paye du personnel est programmée sous la forme d'un modèle de processus indépendant **(1)**. Le clic sur bouton *Paye du personnel* crée un tel processus **(1)**.
On aurait pu également programmer l'édition de texte sous la forme d'un processus indépendant. Cela n'est pas ici nécessaire car sa logique est suffisamment simple pour être programmée entièrement dans les procédures de réaction aux événements des boutons *copier* et *coller*. L'éditeur de texte **(3)** est donc simplement un objet passif qui s'exécute au titre du processus de gestion des événements.

La programmation de cet exemple est un peu fallacieuse, car le mécanisme d'allocation du processeur aux processus est généralement sans partage de temps, avec allocation au processus non logiquement bloqué de plus forte priorité. Dans ce cas, si la paye du personnel n'a pas l'occasion de se bloquer et si c'est un processus de même priorité que la gestion des événements, l'éditeur de texte ne pourra pas s'exécuter. Ici cela marche car on utilise **`sleep()`** dans la boucle du programme de paye. Dans un cas réel, il faudrait lancer la paye du personnel avec une priorité plus faible que la gestion des événements, en remplaçant **(1)** par :

```
PayeDuPersonnel p= new PayeDuPersonnel();
p.setPriority(Thread.currentThread().getPriority() - 1);
p.start();
```


© IFSIC - Université de Rennes 1 - 2003

```
class FenetreDeControle extends Frame {

Button boutonPaye=    new Button("Paye du personnel");
class ActionPaye implements ActionListener {
  public synchronized void actionPerformed(ActionEvent e) {
    (new PayeDuPersonnel()).start(); // lance la paye du personnel
  }
}

Button boutonEdit=    new Button("Edit");
class ActionEdit implements ActionListener {
  public synchronized void actionPerformed(ActionEvent e) {
    new Edit(); // cree une edition de texte
  }
}

Button boutonQuitter=   new Button("Quitter");
class ActionQuitter implements ActionListener {
  public synchronized void actionPerformed(ActionEvent e)
       {System.exit(0);}
}

public FenetreDeControle() {
  super("CONTROLE");
  Placement.p(this,boutonPaye,    0,0,1,1);
  Placement.p(this,boutonEdit,    0,1,1,1);
  Placement.p(this,boutonQuitter, 0,2,1,1);
  boutonPaye.addActionListener(new ActionPaye());
  boutonEdit.addActionListener(new ActionEdit());
  boutonQuitter.addActionListener(new ActionQuitter());
  pack(); setVisible(true);
}
}
```

1

```
class PayeDuPersonnel extends Thread {

class FenetrePaye extends Frame {
  TextField txt= new TextField(40);
  public FenetrePaye() {
    super("PAYE DU PERSONNEL");
    Placement.p(this, txt, 0,0,1,1);
    pack(); setVisible(true);
  }
}
FenetrePaye f= new FenetrePaye();}

public void run() {
  for (int i=100; i>0; i--) {
     f.txt.setText("encore " + i + " secondes avant la paye");
     try{sleep(1000);} catch(Exception e){};
  }
  f.dispose();
}
}
```

2

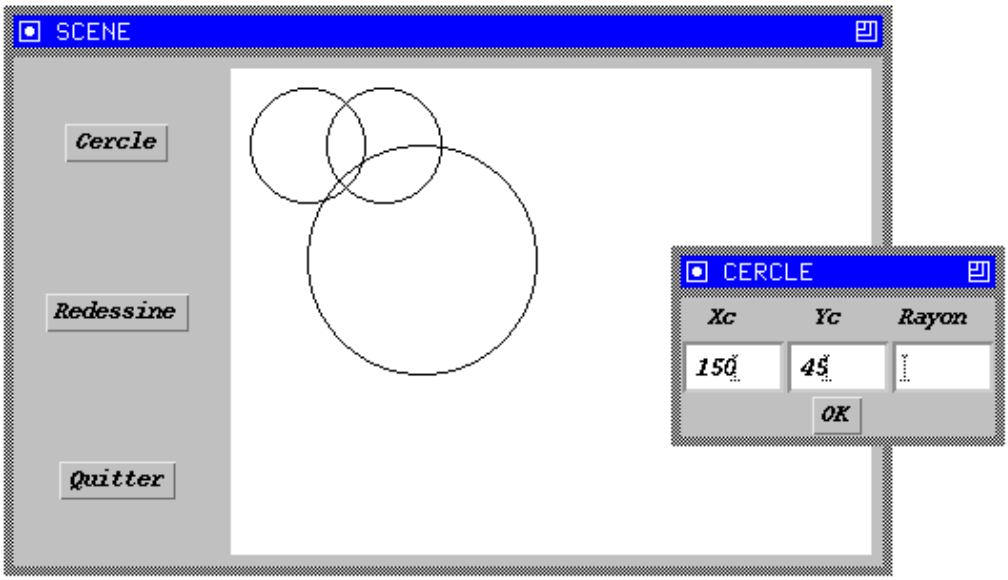
© IFSIC - Université de Rennes 1 - 2005

```
class Edit {                    (3)

class FenetreEdition extends Frame {

  TextArea page= new TextArea(10,80);

  String selection;

  Button boutonCopier=    new Button("copier");
  class ActionCopier implements ActionListener {
    public synchronized void actionPerformed(ActionEvent e) {
      selection=page.getSelectedText();
    }
  }
  Button boutonColler=    new Button("coller");
  class ActionColler implements ActionListener {
    public synchronized void actionPerformed(ActionEvent e) {
     int i = page.getSelectionStart();
     int j = page.getSelectionEnd();
     page.replaceRange(selection,i,j);
    }
  }
  Button boutonFermer=   new Button("fermer");
  class ActionFermer implements ActionListener {
    public synchronized void actionPerformed(ActionEvent e) {
            FenetreEdition.this.dispose();
    }
  }

  public FenetreEdition() {
    super("EDIT");
    Placement.p(this, boutonCopier, 0,0,1,1);
    Placement.p(this, boutonColler, 1,0,1,1);
    Placement.p(this, boutonFermer, 2,0,1,1);
    Placement.p(this, page, 0,1,3,1);
    boutonCopier.addActionListener(new ActionCopier());
    boutonColler.addActionListener(new ActionColler());
    boutonFermer.addActionListener(new ActionFermer());
    pack(); setVisible(true);
  }
}

FenetreEdition f = new FenetreEdition();

}


public class TestInteract {
  static public void main(String[] x) {
      new FenetreDeControle();
}
}
```


© IFSIC - Université de Rennes 1 - 2003

Structure de contrôle plus riche

L'exemple suivant illustre comment l'usage de processus peut aider à structurer agréablement une application.
Il s'agit d'un schéma assez fréquent : l'application permet de construire une "Scène" formée de cercles de tailles et de coordonnées variées. La scène se construit au moyen d'une fenêtre "SCENE" illustrée ci-dessous. Un clic sur *Cercle* provoque la création d'un cercle et son ajout à la scène. La scène est de plus visualisée en permanence sur une zone de dessin de la fenêtre "SCENE".



Un clic sur *Cercle* provoque, en **(2)**, la création et l'ajout d'un cercle.
La création d'un cercle, en **(5)**, est faite par un dialogue (moderne) au moyen d'une fenêtre "CERCLE" par laquelle l'opérateur fournit les coordonnées du centre et le rayon, puis valide la saisie par *OK*.
Mais pour que cette programmation élégante soit possible, il est nécessaire que l'activité qui crée le cercle et l'ajoute à la scène puisse attendre la fin de la saisie.
Or une telle activité, susceptible d'attendre sans perdre son état de contrôle, c'est-à-dire capable de repartir de là où elle en est, c'est justement un processus. Dans cet exemple, la situation d'attente est en **(4)** par suite de l'appel depuis **(5)** par suite de l'appel depuis **(2)** par suite de l'appel depuis **(1)**. Il est impératif que ce soit un processus différent du processus (unique) de gestion des événements qui d'ordinaire se charge de l'exécution des procédures de réaction aux événements, sinon il pourrait attendre longtemps car il attendrait d'être réveillé par lui même, puisque c'est lui qui est censé exécuter la procédure de réaction au bouton *OK*. Il faut donc que la réaction au bouton *Cercle* crée (ou active) un processus indépendant. C'est ce qui est fait en **(1)** : la classe **`P`** est un modèle de processus dont le programme principal appelle **`cercle()`**. La procédure de réaction **`actionPerformed`** crée et lance un tel processus.

Pour réaliser l'attente, on a utilisé le dialogue du cercle comme un moniteur : en **(4)** la création du dialogue attend par **`wait()`** et en **(3)** le clic sur *OK* provoque le réveil par **notify()**.

© IFSIC - Université de Rennes 1 - 2005

```
class Scene extends ListPrcr {

class FenetreDeScene extends Frame {

  Button boutonCercle = new Button("Cercle");
  class ActionCercle  implements ActionListener {

    private class P extends Thread { public void run() {cercle();} }
    public synchronized void actionPerformed(ActionEvent e) {new P().start();}
  }                                                                          (1)

  Button boutonRedessine = new Button("Redessine");
  class ActionRedessine implements ActionListener {
    public synchronized void actionPerformed(ActionEvent e) {dessine();}
  }

  Button boutonQuitter = new Button("Quitter");
  class ActionQuitter implements ActionListener {
    public synchronized void actionPerformed(ActionEvent e) {System.exit(0);}
  }

  Panel dessin= new Panel(); // zone de dessin

  public FenetreDeScene() { super("SCENE");
    Placement.p(this,boutonCercle,   1,1,1,1);
    Placement.p(this,boutonRedessine,1,2,1,1);
    Placement.p(this,boutonQuitter,  1,3,1,1);
    Placement.p(this,dessin,2,1,1,3,GridBagConstraints.CENTER,5,5,5,5,
                                          10,10,GridBagConstraints.BOTH);
    boutonCercle.addActionListener(new ActionCercle());
    boutonRedessine.addActionListener(new ActionRedessine());
    boutonQuitter.addActionListener(new ActionQuitter());
    dessin.setBackground(Color.white); pack(); setVisible(true);
  }
  public void paint(Graphics g) {dessine();};
}

FenetreDeScene f = new FenetreDeScene();

public void cercle()  {ajoutEnFin(new Cercle()); dessine();}   (2)

public void dessine() {
  Graphics g = f.dessin.getGraphics();; // graphique de dessin
  g.clearRect(0,0,2000,1000);
  debut();
  while(! estEnFin()) {((Cercle) eltCourant()).dessine(g); avance();};
}
}
```


© IFSIC - Université de Rennes 1 - 2003

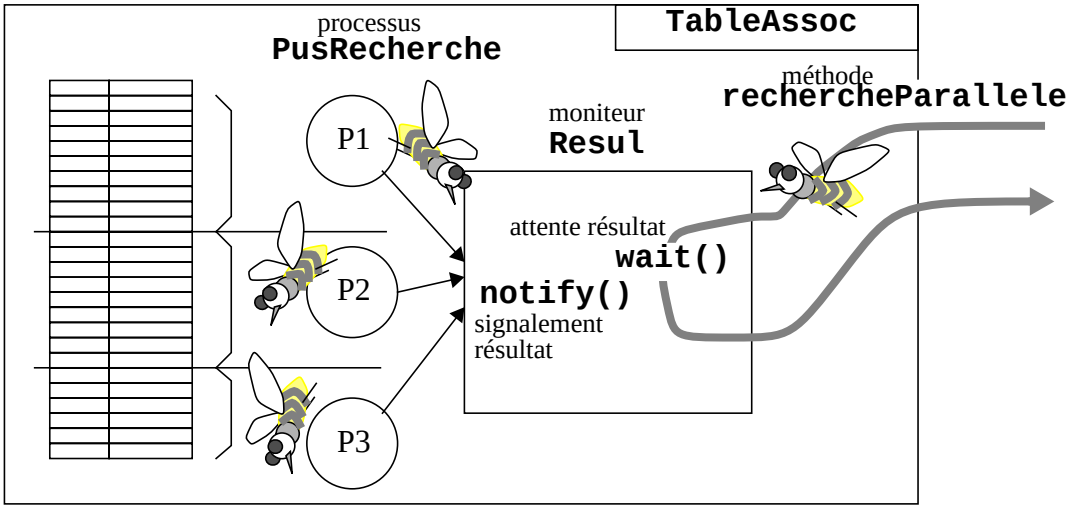
```
class Cercle {

public int Xc; public int Yc; public int rayon;

class DialogueDeCercle extends Frame {
  TextField txtXcentre= new TextField(5);
  TextField txtYcentre= new TextField(5);
  TextField txtRayon  = new TextField(5);

  Button ok=   new Button("OK");
  class ActionOk implements ActionListener {
    public synchronized void actionPerformed(ActionEvent e) {
     reprendre();
    }
  }                                                    3

  public DialogueDeCercle() {
    super("CERCLE");
    Placement.p(this, new Label("Xc"), 0,0,1,1);
    Placement.p(this, txtXcentre,      0,1,1,1);
    Placement.p(this, new Label("Yc"), 1,0,1,1);
    Placement.p(this, txtYcentre,      1,1,1,1);
    Placement.p(this, new Label("Rayon"), 2,0,1,1);
    Placement.p(this, txtRayon,           2,1,1,1);
    Placement.p(this, ok, 1,2,1,1);
    ok.addActionListener(new ActionOk()); pack(); setVisible(true);

    attendre();    4
  }

  public synchronized void reprendre() { notify();}
  public synchronized void attendre() {
    try {wait();} catch(InterruptedException e){};
  }
}

public Cercle() { // constructeur, saisie du cercle par dialogue
  DialogueDeCercle dial= new DialogueDeCercle();
  Xc=Integer.parseInt(dial.txtXcentre.getText());
  Yc=Integer.parseInt(dial.txtYcentre.getText());
  rayon=Integer.parseInt(dial.txtRayon.getText());
  dial.dispose();
}                                                      5

public void dessine(Graphics g) {
  g.drawOval(Xc-rayon,Yc-rayon,2*rayon,2*rayon);
}
}


public class TestControl {
static public void main(String[] x) {new Scene();}
}
```


© IFSIC - Université de Rennes 1 - 2005

Une programmation uniquement faite à l'aide de procédures de réactions est beaucoup plus lourde : une telle procédure ne peut attendre, elle est donc obligée de noter "ce qui reste à faire". Dans cet exemple, la procédure de réaction au clic sur *Cercle* doit se limiter à créer le dialogue de saisie du cercle. Mais que faire de ce cercle quand le dialogue l'a construit, lorsque l'opérateur clique *OK* ? Qui aura noté qu'on se trouve dans la situation captée en **(2)** par le contrôle ? Comment sait-on que ce cercle est destiné à être ajouté à cette scène, et dessiné sur sa zone de dessin ? Dans cet exemple, pourtant simple, il faudrait :
1/ Remplacer la méthode **`cercle()`** de la classe **`Scene`** par deux méthodes : **`debutCercle()`** qui crée un dialogue de cercle pour la saisie, et **`finCercle(Cercle cercleSaisi)`**, appelée lors du clic sur *OK* et qui poursuit l'activité au sujet de ce cercle, à savoir l'ajouter à la scène et le dessiner.
2/ Lors de la création du dialogue de cercle, il faut lui révéler la méthode **`finCercle`** associée à cette scène, de sorte qu'elle soit accrochée comme procédure de réaction au clic sur *OK*. On peut faire cela au moyen d'un objet de type **`ActionListener`** membre de la scène que l'on passe en paramètre à l'initialisation du dialogue de cercle.
Cette façon de faire moins claire que l'usage de processus.

Calcul parallèle

L'exemple suivant est un exemple simple de calcul parallèle. Il s'agit d'une recherche dans une table associative. On a prévu de faire la recherche au moyen de plusieurs processus qui se partagent la recherche dans des portions disjointes de la table.



La méthode **`rechercheParallele`** crée, en **(3)**, un moniteur **`resul`** pour se synchroniser sur l'obtention du résultat et, en **(4)**, des processus de type **`PusRecherche`**, chacun initialisé avec les indices de début et de fin de la portion de table qui leur est attribuée, puis en **(5)** elle attend le résultat pour le rendre.
Le modèle de processus **`PusRecherche`** est donné en **(1)**. Si la clé est trouvée, il transmet la valeur associée par la méthode **`delivre`** du moniteur **`resul`**, sinon il le signale par la méthode **`delivreRien`**.
Le type de moniteur **`Resul`** est présenté en **(2)**. La méthode **`delivre`** réveille l'attendeur. La méthode **`delivreRien`** ne réveille l'attendeur que si aucun processus n'a trouvé la clé, ce qui se détecte en comptant le nombre de processus qui n'ont rien trouvé. Pour une clé qui ne figure pas dans la table, il est convenu que le résultat soit **`null`**.

© IFSIC - Université de Rennes 1 - 2003

```
class TableAssoc {
class Paire {
  String cle; String valeur;
  Paire(String c, String v) {cle=c; valeur=v;}
}

Paire[] T; int nbAssoc;

public TableAssoc(int tailleMax) { T= new Paire[tailleMax]; nbAssoc=0;}

public void associer(String c, String v) {
  T[nbAssoc]= new Paire(c,v); nbAssoc++;
}
```

```
class PusRecherche extends Thread {                                    (1)
  int debut; int fin; String cle;
  PusRecherche(int d, int f, String c) {debut=d; fin=f; cle=c;}
  public void run() {
    for (int i=debut; i<fin; i++) {
      if (cle.equals(T[i].cle)) { resul.delivre(T[i].valeur); return;}
    };
    resul.delivreRien();
  }
}
```

```
class Resul { // moniteur de delivrance du resultat de recherche
  int nbPus; // nombre de processus en cours sur la recherche
  String valeurTrouvee;
  Resul(int nbp) { nbPus = nbp; valeurTrouvee = null;}
  synchronized void delivre(String v) {                                (2)
    valeurTrouvee = v; notify();
  }
  synchronized void delivreRien() {
    nbPus--; if(nbPus==0) {notify();}
  }
  synchronized String attend() {
    try {wait();} catch(InterruptedException e){}; return valeurTrouvee;
  }
}

Resul resul;
```

```
public String rechercheParallele(int nbPus, String cle) {
  int quotaDeBase=nbAssoc/nbPus; int residu=nbAssoc%nbPus;
  resul= new Resul(nbPus);                                             (3)
  int debut=0;
  for (int i=1; i<=nbPus; i++) {
    int fin = debut + quotaDeBase + (residu>0?1:0);
    new PusRecherche(debut,fin,cle).start();                           (4)
    residu--; debut=fin;
  };
  return (resul.attend());                                             (5)
}
}
```


© IFSIC - Université de Rennes 1 - 2005

```
public class CalculPara {
public static void main (String[] x) {
  TableAssoc mineralogique = new TableAssoc(1000);
  mineralogique.associer("1234 BZ 35", "toto");
  mineralogique.associer("1235 BZ 35", "tata");
  mineralogique.associer("1236 BZ 35", "titi");
  mineralogique.associer("1237 BZ 35", "jojo");
  mineralogique.associer("1238 BZ 35", "herve");
  mineralogique.associer("1239 BZ 35", "jacques");
  mineralogique.associer("1240 BZ 35", "louis");
  mineralogique.associer("1241 BZ 35", "gerard");
  mineralogique.associer("1242 BZ 35", "jean");
  mineralogique.associer("1243 BZ 35", "simon");
  mineralogique.associer("1244 BZ 35", "jules");
  System.out.println (mineralogique.rechercheParallele(3,"1241 BZ 35"));
  System.out.println (mineralogique.rechercheParallele(3,"3456 BZ 44"));
}
}
```

8.5 Arrêt d'un processus

On a parfois besoin de forcer la terminaison d'un processus depuis l'exécution d'un autre processus. Les premières versions de Java disposent pour cela de la primitive **`stop()`**. L'exécution de ***p*`.stop()`** provoque la fin du processus ***p*** quelque soit l'état dans lequel il se trouve.
Les versions plus récentes de Java ont déclaré cette primitive ***obsolète*** (*deprecated* en anglais). C'est à juste titre, car bien que cette primitive soit pratique, elle conduit facilement à des programmes erronés. Par exemple, si un processus est détruit alors qu'il est en cours d'exécution d'un moniteur, il laisse sans doute ce moniteur dans un état incohérent.
Une autre technique, plus sûre, pour arrêter un processus consiste à positionner une variable dans un état signifiant la volonté d'arrêter le processus. Cette variable est testée par le processus, en des points convenables de sa programmation et il s'arrête alors de lui-même s'il trouve cette variable dans cet état.
Pour rendre ce principe plus systématique, on peut définir une classe de "processus stoppable" :

```
class StoppableThread extends Thread {
    private boolean stop;
    public StoppableThread(){stop=false;}
    public synchronized void ordreStop() {stop=true;}
    public synchronized boolean testeStop() {
        return stop;
    }
}
```

Pour rédiger un "processus stoppable" il suffit d'hériter de **`StoppableThread`**. Les autres processus disposent de la méthode ***p*`.ordreStop()`** pour arrêter le processus stoppable ***p***, et ***p*** lui-même dispose de **`testeStop()`** pour tester s'il doit s'arrêter.

© IFSIC - Université de Rennes 1 - 2003

Exercice 11

Tampon à N places

Programmer un tampon capable de contenir **n** caractères auquel s'adressent des processus soit pour produire un caractère, soit pour consommer un caractère. Ce tampon se comporte comme un moniteur et fait attendre toute consommation lorsque le tampon est vide et fait attendre toute production lorsque le tampon est saturé. Les caractères sont consommés dans l'ordre de leur production.

Sa programmation sera réalisée par la classe **TamponNplaces** dont voici le schéma :

```
class TamponNPlaces {
// moniteur tampon producteur-consommateur de caractères à n places
  private char[] T;
  private int nbElements;
  private int tete;

  public TamponNPlaces(int n){
  // tampon vide de capacité n caractères
    ...
  }

  public synchronized void produire(char c){
  // effet : attend qu'il y ait la place pour conserver un caractère
  // et rajoute c en queue de this
    ...
}

  public synchronized char consommer(){
  // effet : attend qu'il y ait au moins un caractère dans this
  // et consomme le plus ancien caractère présent
  // résultat : le caractère consommé
    ...
}
```

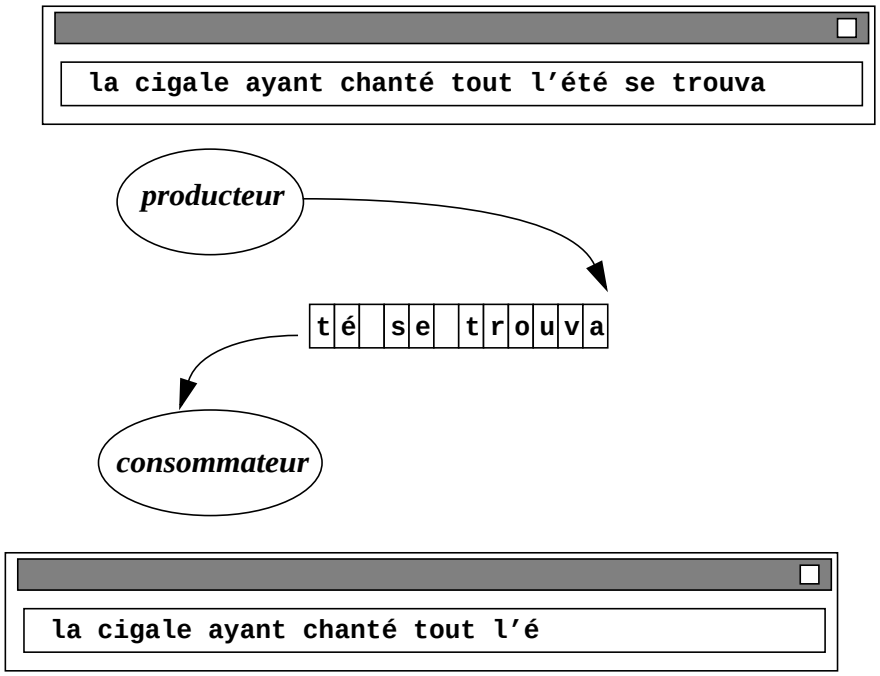
La mise en œuvre proposée, classique, utilise un tableau **T** de **n** caractères pour conserver les caractères. La variable **nbElements** indique le nombre d'éléments présents dans le tampon. Pour éviter des mouvements de données coûteux, la zone occupée dans ce tableau est gérée circulairement :
les caractères présents dans le tampon sont rangés entre l'indice **tete** et l'indice (**tete+nbElements**) ***modulo*** **n**. L'indice **tete** repère le plus ancien caractère, l'indice (**tete+nbElements**) ***modulo*** **n** est la "queue" du tampon, c'est-à-dire l'emplacement où ranger le prochain caractère produit.

© IFSIC - Université de Rennes 1 - 2005

Test du tampon

Rédiger un programme **`TestTampon`** qui réalise la démonstration suivante :
Un processus **`ProducteurDeChaine`** produit un texte, par exemple **`"la cigale ayant chanté tout l'été se trouva fort dépourvue quand la bise fut venue."`**, caractère par caractère, avec un rythme fourni par un ***tirage aléatoire d'une durée d'attente comprise entre 0 et 0,5 seconde*** entre chaque production de caractère.

Un processus **`ConsommateurDeChaine`** consomme le texte produit, caractère par caractère, avec un rythme fourni par un ***tirage aléatoire d'une durée d'attente comprise entre 0 et 1 seconde*** entre chaque consommation de caractère.
Pour visualiser le comportement des processus, le producteur et le consommateur ont une fenêtre (**`Frame`**) contenant une zone de texte (**`TextField`**) dans lequel ils affichent les caractères produits et consommés.



© IFSIC - Université de Rennes 1 - 2003

Exercice 12

Tri parallèle

On se propose de programmer une trieuse “systolique” qui trie des données présentées séquentiellement, en un temps proportionnel au nombre de données.

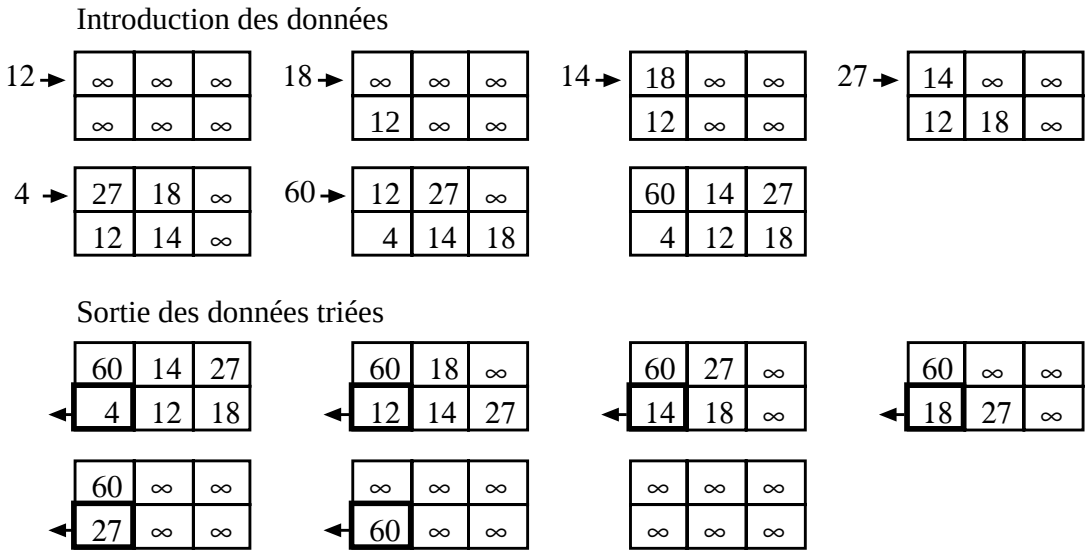
Principe de fonctionnement

La trieuse est constituée d’emplacements, comportant chacun deux registres appelés **`min`** et **`max`**. Tous les registres sont initialisés avec la valeur **`infinie`**. Ensuite le tri se déroule en deux phases :

1) Introduction des données : les données des registres **`max`** sont décalées vers la droite en introduisant une nouvelle donnée dans l’emplacement de gauche et les données **`max`** et **`min`** de chaque emplacement sont échangées si **`max<min`**.

2) Sortie des valeurs triées : la donnée **`min`** de l’emplacement de gauche est lue, les données des registres **`min`** sont décalées vers la gauche en introduisant la valeur **`infinie`** dans l’emplacement de droite et les données **`max`** et **`min`** de chaque emplacement sont échangées si **`max<min`**.

Exemple : tri de la liste de données 12, 18, 14, 27, 4, 60.



1 - Programmer cet algorithme parallèle au moyen d’un processus par emplacement, en utilisant des moniteurs tampons à une place pour le transfert d’information entre emplacements voisins. On prendra un nombre fixe d’emplacements (constante identifiée par **`N`**).

© IFSIC - Université de Rennes 1 - 2005

2 - Bien que conceptuellement intéressant comme exercice de programmation, ce tri parallèle n'offre que peu d'intérêt avec un langage de programmation comme Java qui n'est pas vraiment adapté au "calcul parallèle". Cependant on peut lui trouver un intérêt en tant que "simulateur" pour voir fonctionner un algorithme parallèle. C'est ce qu'on se propose de faire maintenant.

Modifier le programme précédent pour visualiser le comportement de l'agorithme. À chaque emplacement est associée la visualisation de ses registres **`min`** et **`max`**. Les valeurs triées viennent s'inscrire dans une zone de texte à gauche.

Pour pouvoir suivre la visualisation animée du fonctionnement, il faut que chaque emplacement visualise ses registres pendant au moins 0.5 secondes avant et après décision d'échanger ou non **`min`** et **`max`**.

© IFSIC - Université de Rennes 1 - 2003

9 Documents HTML actifs : `Applet`

Une ***applet*** est un programme Java qui peut être référencé depuis un document HTML. Elle est destinée à être téléchargée et exécutée par le navigateur Web lorsque ce document HTML est sollicité.

Pour programmer une applet, il suffit de dériver la classe **`Applet`** définie dans le paquetage **`java.applet`**.

```
class Applet extends Panel {

                    méthodes à définir
void init();
void start();
void paint(Graphics g);
void stop();
...

                    méthodes outils (à utiliser)
void repaint();
URL getCodeBase();
URL getDocumentBase();
AudioClip getAudioClip(URL urlAudioClip);
AudioClip getAudioClip(URL repertoire, String nom);
Image getImage(URL urlImage);
Image getImage(URL repertoire, String nom);

...
}
```

La programmation d'une applet se fait en écrivant les méthodes **`init()`**, **`start()`**, **`stop()`** et **`paint()`**.

La méthode **`init()`** est appelée juste après la création de l'applet et permet de réaliser les initialisations.

La méthode **`paint()`**, héritée de **`Panel`**, est sollicitée chaque fois que le **`Panel`** est découvert, permet de programmer les effets visuels. Elle reçoit en paramètre un contexte graphique **`g`**, objet de classe **`Graphics`**, sur lequel on peut appliquer des opérations de dessin. Une autre méthode de la classe **`Panel`**, **`repaint()`**, sans paramètre, peut être appelée pour forcer le rafraîchissement de la zone graphique de l'applet lorsque l'affichage doit être modifié. Cette méthode appelle **`paint(Graphics g)`** avec le bon paramètre **`g`**.

La méthode **`start()`** est appelée chaque fois que le navigateur ouvre le document (arrivée sur la page, désiconification avec certains navigateurs). Elle sert à lancer ou relancer les traitements associés au document, tels que des animations ou des effets sonores, généralement exécutés au moyen de processus séparés.

© IFSIC - Université de Rennes 1 - 2005

La méthode **stop()** est appelée chaque fois que le navigateur quitte le document . Elle sert à arrêter les traitements parallèles associés au document, tels que des animations ou des effets sonores.

Une applet possède automatiquement, par héritage, un **Panel** qui permet des affichages graphiques au sein du document HTML.

Voici quelques méthodes de dessin offertes par la classe **Graphics** :

```
class Graphics {

void drawLine(int x1,int y1,int x2,int y2)   dessine un segment
void drawRect(int x,int y,int w,int h)       dessine un rectangle
void drawOval(int x,int y,int w,int h)       dessine une ellipse
void drawString(String s, int x,int y)       dessine un texte

void setColor(Color c)
Color getColor()          assigne/obtient la couleur courante de dessin

void setFont(Font f)
Font getFont()            assigne/obtient la fonte courante de dessin de texte

boolean drawImage(Image im, int x,int y, ImageObserver o)
boolean drawImage(Image im, int x,int y, int w,int h,
                                                ImageObserver o)
```

affiche l'image **im** dans le rectangle (**x**,**y**,**w**,**h**). Le paramètre **o** est utile pour des images longues à charger. La méthode **drawImage()** peut être appliquée alors que l'image **im** n'est pas totalement chargée, et **o** est l'entité qui doit être prévenue lorsque l'affichage est achevé. Usuellement **o** est l'objet de classe **Component** dans lequel l'image est affichée.

```
...
}
```

L'insertion de l'applet dans le document HTML se fait au moyen de balises :

```
<APPLET CODE=xxx.class WIDTH=250 HEIGHT=150> </APPLET>
```

La rubrique **CODE** indique le nom du fichier **.class** contenant l'applet. **WIDTH** et **HEIGHT** fixent la largeur et la hauteur, en pixels, de la zone graphique au sein du document.

L'exemple suivant illustre une applet qui affiche un petit chronomètre qui s'incrémente chaque seconde :

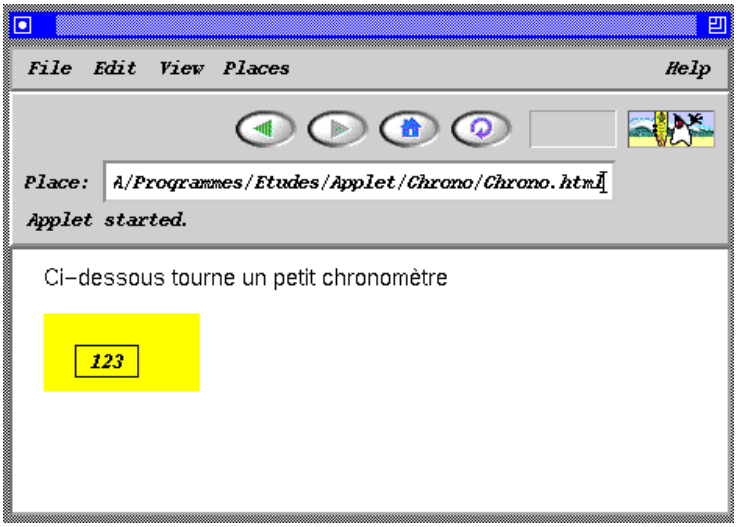
© IFSIC - Université de Rennes 1 - 2003

```
import java.applet.*; import java.awt.*;

public class Chrono extends Applet {

int date;

   class IncrementDate extends StoppableThread {
    public void run() {
      while (!testeStop()) {
      try {sleep(1000);} catch(InterruptedException e) {}
        date++; Chrono.this.repaint();
      }
    }
   }
IncrementDate incr;

public void init () {date = 0;}
public void start () {
  incr=new IncrementDate(); incr.start();
}
public void stop () {incr.ordreStop();}
public void paint (Graphics g) {
   setBackground(Color.yellow); g.drawRect(20,20,40,20);
   g.drawString(String.valueOf(date),30,35);
}
}
```

Un document HTML activant cette applet pourrait être :

```
<HTML>
<HEAD> <TITLE> CHRONO </TITLE> </HEAD>
<BODY>
<P> Ci-dessous tourne un petit chronomètre </P>
<APPLET CODE= Chrono.class WIDTH=100 HEIGHT=50> </APPLET>
</BODY>
</HTML>
```

La visualisation du document donne :



© IFSIC - Université de Rennes 1 - 2005

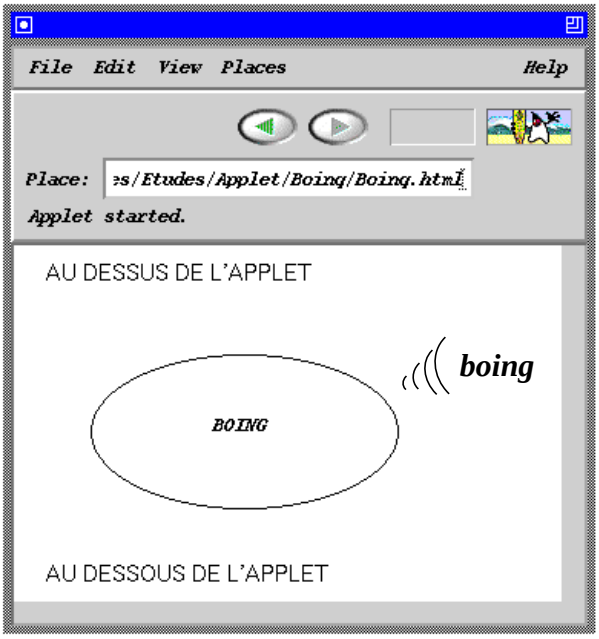
La classe **`Applet`** permet l'affichage d'images (fichiers **`.gif`** ou **`.jpg`**) et l'émission de sons (fichiers **`.au`**).

La méthode **`Image getImage(...)`** rend une représentation interne d'une image contenue dans un fichier.
On peut afficher une image **`im`** par **`g.drawImage(im,....)`** , **`g`** étant le contexte graphique associé à l'applet.

La méthode **`AudioClip getAudioClip(...)`** rend une représentation interne d'un enregistrement sonore contenu dans un fichier.
On peut jouer un enregistement sonore **`m`** par **`m.play()`**.

Exercice 13

Programmer une applet dont le comportement est illustré ci-dessous :



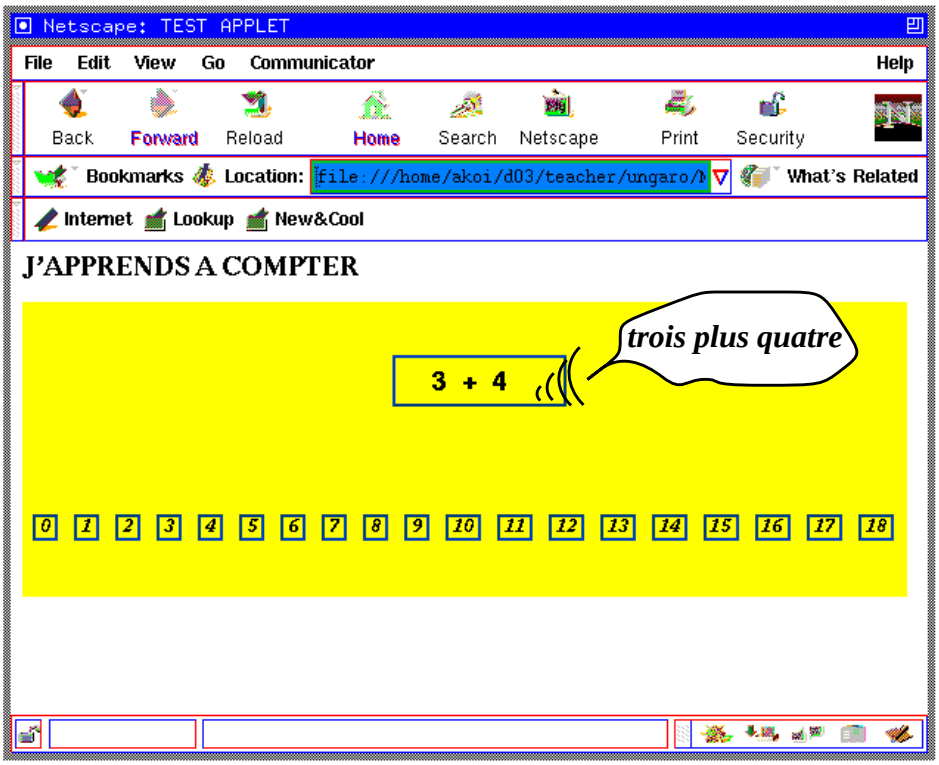
Périodiquement, l'applet affiche une ellipse aplatie pendant 0,5 seconde puis une ellipse plus enflée tout en prononçant "boing" au moyen d'un fichier de son **`boing.au`**.

© IFSIC - Université de Rennes 1 - 2003

Exercice 14

Applet “j’apprends à compter”

On se propose de programmer une applet pour apprendre à compter. Le programme tire deux nombres i et j au hasard compris entre 0 et 9 et pose vocalement la question “i plus j”.



L’utilisateur peut donner la réponse en cliquant sur un des 19 boutons prévus à cet effet.

- Si la réponse est correcte, le programme répond “exact”, sinon il répond “non, la réponse est i+j”.
- Si l’utilisateur ne donne pas de réponse au terme d’un délai de 5 secondes, le programme donne la bonne réponse “la réponse est i+j”.

Toutes les réponses du programme sont données oralement.

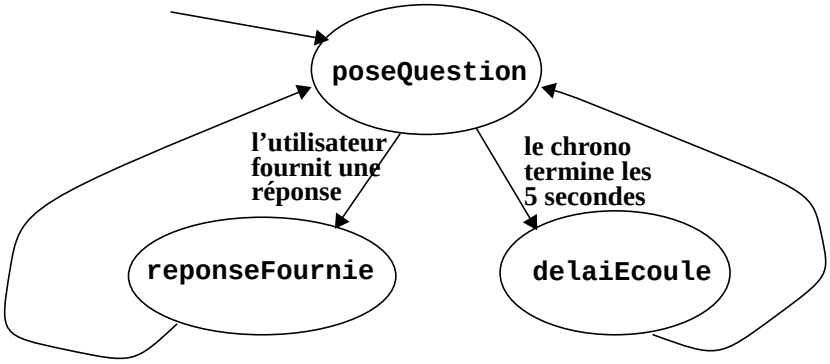
On dispose des fichiers audio :

`n0.au`, **`n1.au`**, ... **`n18.au`** pour prononcer les nombres,
`plus.au`, **`exact.au`**, **`non.au`**, **`laReponseEst.au`** pour former les phrases.

© IFSIC - Université de Rennes 1 - 2005

Principe de fonctionnement

Le fonctionnement est basé sur un automate à trois états :



Dans l'état **`poseQuestion`**, un processus **`Questionneur`** pose une question, crée et lance un chronomètre puis attend.

Le chronomètre est un processus **`Chrono`** qui attend 5 secondes.
S'il a l'occasion de terminer son attente, il fait passer l'état à **`delaiEcoule`** et réveille le questionneur.
Si avant le délai de 5 secondes l'utilisateur fournit une réponse, le questionneur est réveillé dans l'état **`reponseFournie`** : il analyse la réponse et détruit le chronomètre en cours.

© IFSIC - Université de Rennes 1 - 2003

10 Communications réseau : net

Le paquetage **java.net** offre les moyens de communication entre machines à travers les réseaux, et notamment à travers Internet. Il offre :

- la manipulation des adresses Internet grâce à la classe **InetAddress**,
- les communications de bas niveau grâce aux classes **DatagramPacket** et **DatagramSocket**,
- les communications en mode connecté grâce aux classes **Socket** et **ServerSocket**,
- l'accès de type fichier désigné par URL grâce aux classes **URL** et **URLConnection**.

10.1 Adresses Internet

Les machines connectées sur Internet sont identifiées par une adresse unique de 32 bits (en 1999). Ces adresses sont représentées en Java par la classe **InetAddress** :

```
class InetAddress {
static InetAddress getLocalHost()
                            throws UnknownHostException;;
static InetAddress getByName(String Machine)
                            throws UnknownHostException;
String getHostName();
}
```

La méthode **getLocalHost()** rend en résultat l'adresse de la machine locale (celle où s'exécute couramment le programme). La méthode **getByName(String Machine)** rend en résultat l'adresse associée à la chaîne de caractères **Machine**. Cette chaîne peut être soit la représentation d'une adresse de 32 bits sous forme de 4 nombres compris entre 0 et 255, par exemple **"131.254.52.9"**, soit un nom en clair **"poseidon.ifsic.univ-rennes1.fr"**. La méthode **getHostName()** retourne le nom de la machine associée à une adresse.

10.2 Communications de bas niveau

Il s'agit de transmissions de paquets individuels, en mode non connecté. Ces transmissions sont peu fiables : elles ne garantissent pas la réception des paquets et n'assurent pas que l'ordre d'arrivée soit le même que l'ordre d'émission.

Les paquets sont représentés par des objets de classe **DatagramPacket** :

```
class DatagramPacket {
DatagramPacket(byte[] buf, int lng);
DatagramPacket(byte[] buf, InetAddress adr, int port);
InetAddress getAddress();
byte[] getData;
int getLength()
int getPort()
}
```


© IFSIC - Université de Rennes 1 - 2005

`DatagramPacket(byte[] buf, int lng)` : ce premier constructeur crée un objet destiné à recevoir un paquet.

`DatagramPacket(byte[] buf, InetAddress adr, int port)` : ce constructeur spécifie un paquet destiné à être émis vers le port **`port`** du site d'adresse **`adr`**.

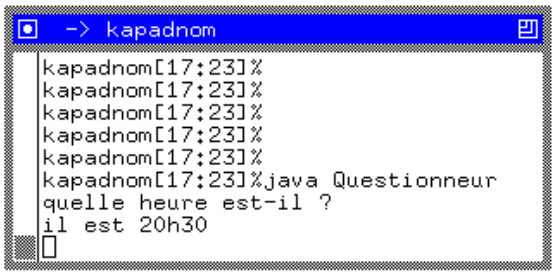
L'émission et la réception de paquets se font par l'intermédiaire d'un socket, représenté en Java par un objet de classe **`DatagramSocket`** :

```
class DatagramSocket {
DatagramSocket() throws SocketException;
DatagramSocket(int port) throws SocketException;
int getLocalPort();
void send(DatagramPacket p);
void receive(DatagramPacket p);
}
```

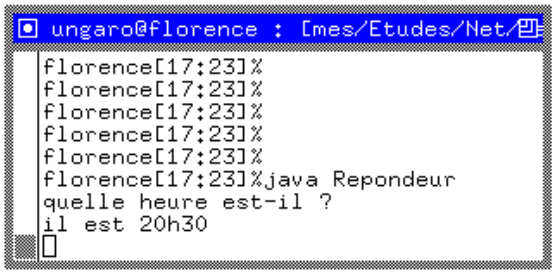
Le constructeur **`DatagramSocket()`**crée un socket associé à un numéro de port disponible décidé par le système. Le constructeur **`DatagramSocket(int port)`** impose le numéro de port indiqué. La méthode **`getLocalPort()`** permet de connaître le numéro du port associé.
La méthode **`send(DatagramPacket p)`** envoie le paquet **`p`**.
La méthode **`receive(DatagramPacket p)`** range le prochain paquet reçu dans l'objet **`p`**.

Les programmes suivants illustrent la communication entre des questionneurs (classe **`Questionneur`**) et un répondeur (classe **`Repondeur`**). Les questionneurs posent des questions. Le répondeur reçoit les questions et y répond. Questions et réponses sont des chaînes de caractères frappées au clavier et validées par passage à la ligne :

site client : **`Questionneur`**



site serveur : **`Repondeur`**



Le site où s'exécute le répondeur s'appelle un ***site serveur*** : son adresse est a priori connue des questionneurs. Dans l'exemple, il s'agit de la machine "**`florence.irisa.fr`**". Un site où s'exécute un questionneur est un ***site client*** : son adresse est initialement inconnue du site serveur. Le serveur en prend connaissance par les paquets qu'il reçoit.

© IFSIC - Université de Rennes 1 - 2003

```
import java.net.*; import es.*;

public class Questionneur {

public static void main(String[] x){
   try {
   InetAddress adrRepondeur =
            InetAddress.getByName("florence.irisa.fr");
   int portRepondeur = 8080;
   DatagramPacket paquetReponse =
                new DatagramPacket(new byte[1000], 1000);

   DatagramSocket socket = new DatagramSocket();

   String question = Lecture.chaine("\n");
   while (!question.equals(".")) {

      // envoit la question
      byte[] strb= question.getBytes();
      byte[] sbuf=new byte[1000];
      for (int i=0;i<strb.length;i++){sbuf[i]=strb[i];};
      DatagramPacket paquetQuestion =
          new DatagramPacket(sbuf,1000,adrRepondeur,portRepondeur);
      socket.send(paquetQuestion);

      // recoit la reponse
      socket.receive(paquetReponse);
      System.out.println(
               new String(paquetReponse.getData()));

      question = Lecture.chaine("\n");
   }
   }catch(Exception e){
   System.out.println("ERREUR"); System.exit(0);
   };
}
}
```


© IFSIC - Université de Rennes 1 - 2005

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


public class Repondeur {

int portRepondeur= 8080;
DatagramSocket socketClient;

byte[] buf = new byte[1000];
DatagramPacket paquetQuestion= new DatagramPacket(buf, buf.length);

InetAddress adrClient ;
int portClient;


public Repondeur() {

  try{ socketClient = new DatagramSocket(portRepondeur);}
  catch(SocketException e){}

  while (true) {
    try{socketClient.receive(paquetQuestion);}
    catch(Exception e){return;};
    adrClient = paquetQuestion.getAddress();
    portClient = paquetQuestion.getPort();
    System.out.println(new String(paquetQuestion.getData()));
    byte[] strb= Lecture.chaine("\n").getBytes();
    byte[] sbuf=new byte[1000];
    for (int i=0;i<strb.length;i++){sbuf[i]=strb[i];};
    DatagramPacket paquetReponse =
    new DatagramPacket(sbuf,100,adrClient,portClient);
    try {socket.send(paquetReponse);} catch(Exception ex){};
  }
}


public static void main(String[] a) { new Repondeur();}
}
```

Exercice
Modifier la programmation de la classe **`Repondeur`** de façon à ce que la question affichée soit préfixée par l'identification de la machine cliente, sous la forme :
nom de la machine cliente **`:`** ***question***

10.3 Communications en mode connecté

La classe **`Socket`** définit un canal de transmission entre machines. Après avoir créé deux objets de type **`Socket`**, un sur chacun des sites communicants, et après les avoir connectés, les communications se font au moyen de canaux d'entrées/sorties séquentielles de type **`InputStream`** et **`OutputStream`**. Le protocole de communication (TCP) garantit le respect de l'ordre.

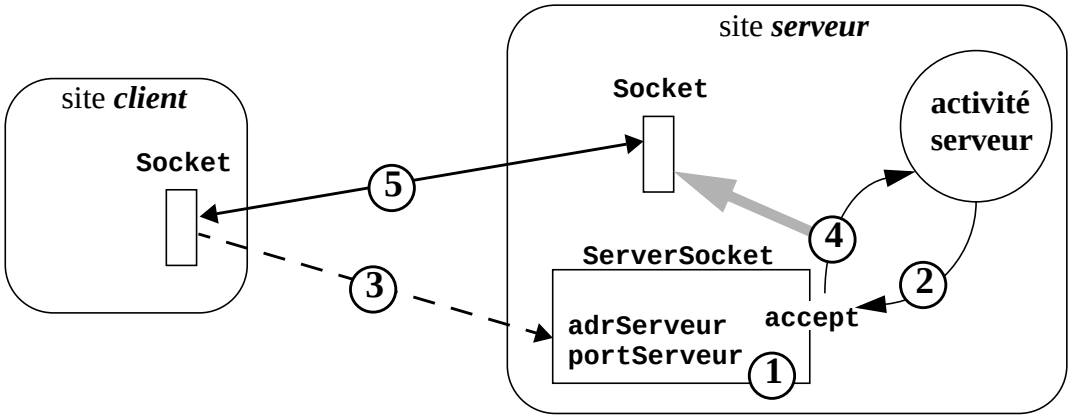
© IFSIC - Université de Rennes 1 - 2003

```
class Socket {
Socket(String host, int port);
Socket(InetAddress adr, int port);
InetAddress getInetAddress();
InputStream getInputStream();
OutputStream getOutputStream();
void close();
}
```

La classe **Socket** ne suffit pas car l'établissement d'une connexion est toujours disymétique : il y a un côté ***serveur***, qui préexiste, dont l'adresse est supposée connue, et qui attend des clients, et un côté ***client*** qui a l'initiative de l'établissement de la connexion. Ceci nécessite une seconde sorte d'objet côté serveur, un **ServerSocket**, dont le rôle est d'être à l'écoute des clients.

```
class ServerSocket {
ServerSocket(int port);
InetAddress getInetAddress();
Socket accept();
void close();
}
```

Le mécanisme d'établissement d'une connexion est illustré sur le schéma suivant :
(**1**) : l'activité serveur crée un **ServerSocket** sur le site serveur.
(**2**) : l'activité serveur appelle la méthode **accept()** de ce **ServerSocket**, ce qui la met en attente d'une manifestation de la part d'un client.
(**3**) : un site client crée un **Socket** en citant l'adresse et le port du serveur. Ceci sollicite le **ServerSocket** associé à cette adresse et ce port. Le **ServerSocket** crée alors un **Socket** sur le site serveur et le connecte à ce client.
(**4**) : La méthode **accept()** termine en rendant ce **Socket**.
(**5**) : Le client et le serveur peuvent alors dialoguer grâce aux flux d'entrée et de sortie offerts par les méthodes **getInputStream()** et **getOutputStream()** de leur **Socket** respectif.



Pour émettre et recevoir sur des objets de type respectivement **OutputSream** et **InputStream**, on dispose des méthodes **void writeChars(String s)** et **char readChar()**.

© IFSIC - Université de Rennes 1 - 2005

L'exemple suivant est la programmation d'un questionneur et d'un répondeur du même type que précédemment.

```
public class Repondeur {

public static void main(String[] arg) {
  try{

    ServerSocket accesRepondeur = new ServerSocket(8082);

    Socket socketClient = accesRepondeur.accept();
    DataInputStream inClient =
              new DataInputStream(socketClient.getInputStream());
    DataOutputStream outClient =
              new DataOutputStream(socketClient.getOutputStream());

    while (true) {
      String question="";
      char c=inClient.readChar();
       while(c!='.'){question=question+c; c=inClient.readChar();};
      System.out.println(question);

      String reponse = Lecture.chaine("\n");
      outClient.writeChars(reponse + '.');
    }
  }
  catch(Exception e){System.out.println("ERREUR"); System.exit(0);}
}
}
```

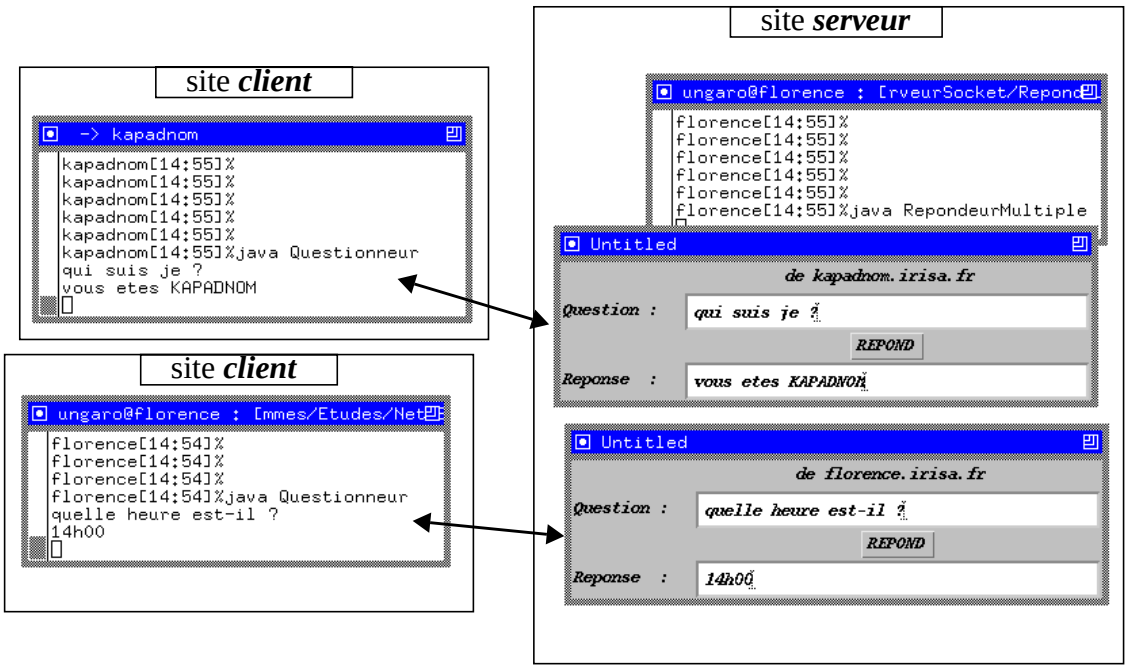
```
public class Questionneur {

public static void main(String[] arg) {
  try{
    Socket socketServeur = new Socket("florence.irisa.fr",8082);
    DataInputStream inServeur=
            new DataInputStream(socketServeur.getInputStream());
    DataOutputStream outServeur =
            new DataOutputStream(socketServeur.getOutputStream());

    while (true) {
      String question = Lecture.chaine("\n");
      outServeur.writeChars(question + '.');

      String reponse="";
      char c=inServeur.readChar();
      while(c!='.'){reponse=reponse+c; c=inServeur.readChar();};
      System.out.println(reponse);
    }
  }
  catch(Exception e){System.out.println("ERREUR"); System.exit(0);};
}
}
```


© IFSIC - Université de Rennes 1 - 2003

Exercice

La programmation précédente du questionneur et du répondeur est simple mais peu illustrative d'une relation entre des clients et un serveur. Le propre des clients est d'être potentiellement nombreux et, dans la mesure où un service est long et interactif, il est souhaitable que les clients soient servis en parallèle.

Programmer un site serveur qui offre un service de répondeur à plusieurs clients simultanément. Sur le site serveur, chaque service répondeur en cours donnera lieu à une fenêtre dotée d'une zone de texte pour afficher la question, une zone de texte pour préparer la réponse et un bouton pour envoyer la réponse :



© IFSIC - Université de Rennes 1 - 2005

10.4 Accès par URL

La classe **URL** permet de faire apparaître comme un fichier séquentiel de type **InputStream** n'importe quelle ressource accessible sur le Web au moyen d'une URL.

```
class URL {
URL(String url);
Socket(InetAddress adr, int port);
String getFile();
String getHost();
InputStream openStream();
URLConnection openConnection();
}
```

Une façon simple de l'utiliser est de créer un objet **URL** au moyen du constructeur avec l'URL en paramètre, sous forme d'une chaîne de caractères de la forme

http://...

puis d'utiliser la méthode **openStream()** pour obtenir un objet de type **InputStream** qui permet de lire le contenu de cette URL.

L'exemple suivant est un programme qui lit au clavier un nom d'URL, supposé désigner un fichier texte, et l'imprime sur le terminal :

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

public class TestUrl {
public static void main(String[] arg) {
  System.out.println("url :");
  String chaineUrl = Lecture.chaine("\n");
  try{
  URL url = new URL(chaineUrl);
  InputStream in = url.openStream();
  int k= in.read();
   while (k!=-1) { System.out.print((char) k); k= in.read(); }
}
 catch(EOFException e) {System.out.println("\ntermine");}
 catch(Exception e) {System.out.println("erreur"); System.exit(0);};
}
}
```

La méthode **openConnection()** rend un objet de type **URLConnection**, que nous ne décrirons pas ici, qui permet des accès plus élaborés à l'URL.

© IFSIC - Université de Rennes 1 - 2003

11 Applications réparties : `rmi`

Le réseau qui relie les machines n'offre à la base que des communications par messages. Cependant dans une application répartie, le schéma le plus fréquent est la sollicitation, par un site client, d'un service offert sur un site serveur. On peut certes tout programmer à l'aide de messages, mais cette programmation est fastidieuse. En effet pour appeler un service correspondant à une procédure de la forme :

```
TypeResultat NomDuService(Type1, ... Typek)
```

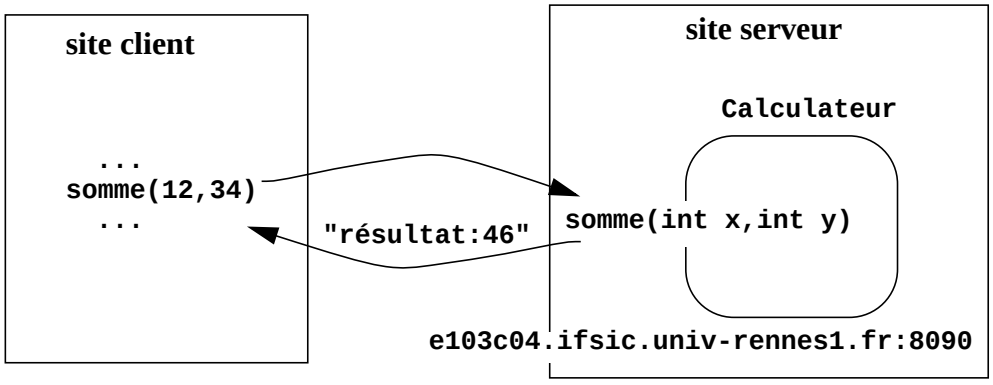
- le site client doit envoyer un message de la forme :
  `CodeService, p1, ... pk`
  puis se mettre en attente du résultat,
- le site serveur doit recevoir le code du service demandé, le décoder (au moyen d'un ***switch*** par exemple), puis recevoir les *k* paramètres en les décodant et en restituant leur types respectifs, appeler la procédure de ce service et renvoyer un message signifiant le résultat.

Non seulement cette programmation n'est pas performante, à cause des codages et des décodages concernant le service, ses paramètres et son résultat, mais elle est confuse et source d'erreurs puisque le compilateur ne peut faire aucun contrôle sur le nombre et le type des paramètres.

L'appel de méthodes d'objets distants permet d'éviter ces inconvénients. Dans un système qui offre ce mécanisme on peut définir des objets "distants" (***remote objects***) qui peuvent résider sur un site et être utilisés depuis un autre, en appelant ses méthodes comme si l'objet était local. Pour le cas d'applications réparties faisant intervenir divers langages (C++, Smalltalk, Eiffel, Java, ...) il existe une norme, appelée CORBA, qui, au moyen de bibliothèques offertes par ces divers langages, permet de pratiquer l'invocation de méthode à distance. Cependant la mise en œuvre en CORBA est assez lourde et lorsqu'on se limite au seul langage Java, le moyen le plus élégant de construire une application répartie est offert par le paquetage `rmi` (***Remote Method Invocation***).

11.1 Mécanisme illustré sur un exemple

Nous prendrons l'exemple simple suivant pour illustrer le mécanisme et les moyens de l'exprimer en Java.



© IFSIC - Université de Rennes 1 - 2005

Un ***site serveur*** définit un objet **`Calculateur`** doté d'une méthode **`somme()`** qui additionne deux entiers passés en paramètre et rend le résultat sous forme d'une chaîne de caractères.

Sur un ***site client*** on peut accéder à cet objet par un ***nom externe*** qui est une chaîne de caractères formée du nom du serveur, d'un numéro de port, et d'un identificateur d'objet, de la forme :
`//e103c04.ifsic.univ-rennes1.fr:8090/additionneur`

Le client obtient la référence de cet objet puis l'utilise directement en appelant la méthode **`somme()`**.

Pour réaliser cela il faut procéder comme suit :

11.1.1 Définition de l'interface d'un objet distant

Le profil des méthodes offertes par un objet distant doit être défini au moyen d'une **`interface`** qui hérite de **`Remote`** :

```
import java.rmi.*;

public interface InterfaceCalculateur extends Remote {
public String somme(int x,int y) throws RemoteException;
}
```

Toute méthode de l'interface doit avoir **`throws RemoteException`** dans son profil.
Les utilisateurs manipuleront un objet distant uniquement par les appels de méthodes définies dans cette interface.

11.1.2 Définition d'une classe qui implémente un type d'objet distant

Un objet distant est destiné à être créé sur un site que l'on appellera le "site serveur", de cet objet. Un tel objet doit être défini au moyen d'une classe qui ***hérite*** de **`UnicastRemoteObject`** et qui ***met en œuvre*** l'interface déclarée pour ce type d'objet distant. **`UnicastRemoteObject`** est une classe de base pour les objets capables d'être référencés à distance.
Dans l'exemple, la classe **`Calculateur`** réalise les objets distants spécifiés par l'interface **`InterfaceCalculateur`** :

```
import java.rmi.*;
import java.rmi.server.*; import java.rmi.registry.*;

class Calculateur extends UnicastRemoteObject
                                implements InterfaceCalculateur {
public ServeurCalculateur() throws RemoteException {...}
public String somme(int x, int y) { return ("resultat : " + (x+y)); }
}
```

Remarque : le constructeur doit avoir **`throws RemoteException`** dans son profil. En revanche, cela n'est pas nécessaire pour les méthodes.

© IFSIC - Université de Rennes 1 - 2003

11.1.3 Initialisation du site serveur et enregistrement d'un objet distant

Le programme suivant, exécuté sur le site serveur, crée un objet **`Calculateur`** et l'associe à un ***nom externe*** afin que le monde extérieur puisse le désigner.

```
class Calculateur {
public static void main(String[] x) {
  try {
    LocateRegistry.createRegistry(8090);
    Calculateur gaston = new Calculateur();
    Naming.bind(
    "//e103c04.ifsic.univ-rennes1.fr:8090/additionneur",
    gaston);
  }
  catch(Exception e) { System.out.println("erreur"); return;};
}
}
```

Il faut également lancer, sur le site serveur, un processus destiné à enregistrer les associations, au titre du port **`8090`** dans cet exemple, entre des adresses externes et des objets que l'on désire rendre accessibles de cette manière. Ceci peut être fait par :
`LocateRegistry.createRegistry(8090)`

Enfin, l'appel : **`Naming.bind(...)`**
associe l'adresse externe **`//e103c04.ifsic.univ-rennes1.fr:8090/additionneur`** à l'objet **`gaston`** de type **`Calculateur`** qui vient d'être créé.

11.1.4 Connexion d'un client à un objet distant connu par son nom externe

Le programme ci-dessous, exécuté sur un site client, obtient la référence à un objet distant de type **`Calculateur`** puis l'utilise en appelant sa méthode **`somme`** :

```
import java.rmi.*; import java.rmi.registry.*; import es.*;

public class ClientCalcul {

public static void main(String[] x){
  try {
    InterfaceCalculateur calc =
     (InterfaceCalculateur) Naming.lookup(
        "//e103c04.ifsic.univ-rennes1.fr:8090/additionneur");
    while (!Lecture.chaine().equals(".")) {
      int x = Lecture.unEntier();
      int y = Lecture.unEntier();
      System.out.println(calc.somme(x,y));
    }
  }
  catch(Exception e){
    System.out.println("erreur"); System.exit(0);
  }
}}
```

L'appel **`Naming.lookup(...)`**
Rend une référence à l'objet connu sous le nom externe
`//e103c04.ifsic.univ-rennes1.fr:8090/additionneur`

© IFSIC - Université de Rennes 1 - 2005

La méthode **`lookup`** rend une référence non typée. Il faut la typer convenablement au moyen d'un "cast".
On peut alors utiliser cette référence comme si l'objet était local. Lors d'une invocation de méthode, le système sous-jacent s'occupe de transformer le passage des paramètres et le rendu de résultat en envoi de messages vers le site serveur et réception de messages depuis le site serveur.

11.1.5 Compilation des souches et des squelettes (`stubs` & `skeletons`)

Ce que nous venons de voir est suffisant en ce qui concerne les programmes sources. Cependant le système nécessite, pour chaque classe d'objet distant, deux classes supplémentaires, la souche (**`stub`**) et le squelette (**`skeleton`**).
Le squelette contient la logique qui s'occupe de recevoir les paramètres et retourner le résultat du côté serveur. La souche contient la logique qui émet les paramètres et récupère les résultats côté client.

L'existence de ces classes est totalement transparente à la programmation, mais cela exige une compilation supplémentaire. Cette compilation se fait au moyen de la commande **`rmic`** (***rmi compiler***) avec pour argument le fichier **`.class`** de la classe qui implémente des objets distants. Il faut donc l'invoquer après compilation du source de cette classe.
Dans notre exemple, après avoir exécuté :

```
javac Calculateur.java
```

on exécute

```
rmic Calculateur
```

ce qui crée les deux fichiers :

```
Calculateur_Skel.class
Calculateur_Stub.class
```

Pour tester l'exemple, il suffit alors de lancer le serveur sur le site prévu, à savoir la machine **`e103c04.ifsic.univ-rennes1.fr`** dans cet exemple, et de lancer autant de clients que l'on veut, sur n'importe quelles machines.

11.2 Communication de références d'objets distants

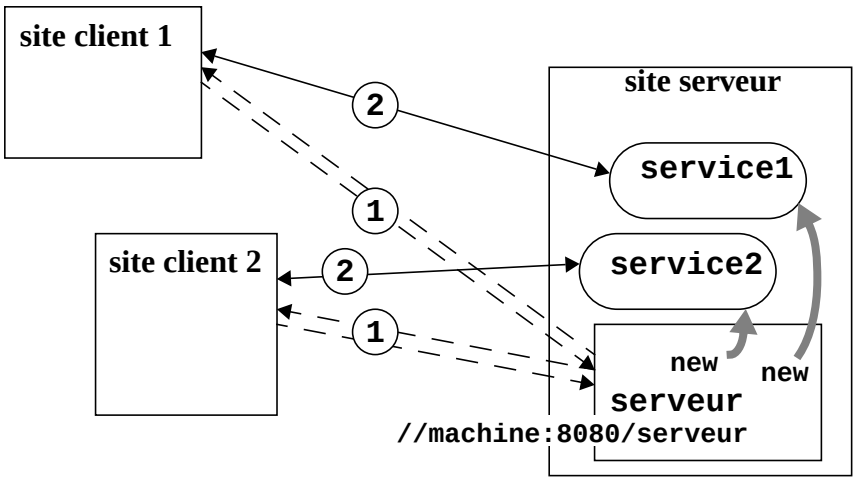
Dans l'exemple précédent nous n'avons qu'un seul objet distant et cet objet est connu par les clients grâce à une adresse externe, donnée sous forme d'une chaîne de caractères.
Dans toute application un peu plus réaliste, la plupart des objets ne sont pas désignés ainsi. L'application crée des objets distants et leurs références sont communiquées en tant que paramètres ou résultats d'invocations de méthodes.

© IFSIC - Université de Rennes 1 - 2003

Les objets distants, c'est-à-dire de classe **`Remote`**, peuvent être passés en paramètre ou rendus en résultat de méthodes, ***y compris de méthodes d'objets distants***. Dans ce cas, c'est la référence à l'objet qui est transmise, comme pour un objet local. La seule différence est que, de façon transparente, lors des transmissions entre sites, ces références sont transmises sous une forme adéquate, sans doute composée de l'adresse du site qui héberge l'objet et d'une référence interne à ce site.

En règle générale, dans une application répartie, seul un objet (ou quelques objets) est connu par un nom externe, un objet "serveur central" de l'application répartie. Cet objet préexistant est le germe d'activités qui ensuite créent d'autres objets distants et se les communiquent par paramètres ou résultats de procédures, de façon tout à fait conventionnelle, comme si ces objets étaient situés sur le même site.

L'exemple suivant illustre un schéma fréquent : un site **client 1** s'adresse à un serveur **`serv`** qui est un objet connu par un nom externe **`"//machine:8080/serveur"`** en appelant une méthode **(1)** qui crée un objet spécifique **`service1`**, dont la référence est rendue en résultat au client. Ensuite, **(2) client 1** dialogue directement avec cet objet **`service1`**. Et ce principe est reproductible pour autant de clients que l'on veut.



11.3 Passage des paramètres et rendu de résultat

Lorsqu'un paramètre ou un résultat de méthode d'objet distant est d'un type de base (scalaire), le passage se fait par valeur.

Lorsqu'un paramètre ou un résultat de méthode d'objet distant est d'un type classe d'objet distant (dérivé de **`Remote`**), c'est la référence à l'objet qui est transmise.

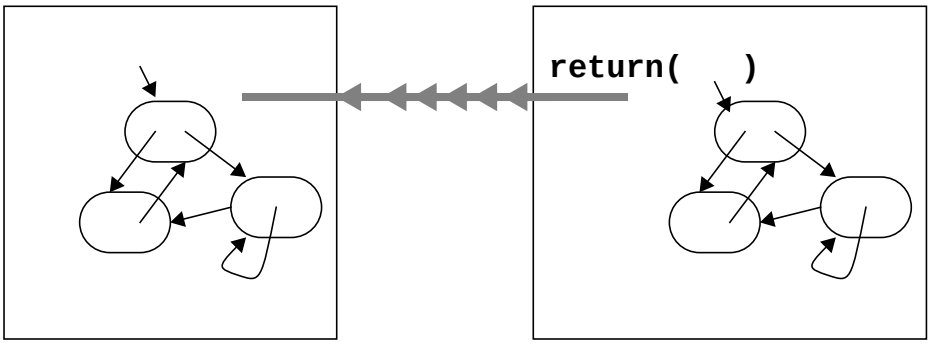
En revanche si le paramètre ou le résultat est d'un type classe non distant, c'est, sous certaines conditions, un clone de l'objet qui est transmis : la valeur de l'objet est transmise, selon un format particulier standardisé, et à l'autre extrémité un nouvel objet est créé avec les valeurs reçues. Cette création est un peu particulière dans la mesure où elle ne sollicite aucun constructeur.

© IFSIC - Université de Rennes 1 - 2005

Pour que le clonage soit possible, il faut cependant que l'objet transmis implémente l'interface **`Serializable`**. Cette interface est en fait une interface vide (il n'y a rien à rédiger de façon standard). Il suffit simplement que la classe des objets que l'on souhaite ainsi transmettre soit déclarée :

```
class UnObjetTransmissible implements Serializable { ...}
```

Le clonage réalisé est astucieux : si l'on a un réseau d'objets **`Serializable`** qui se référencent mutuellement, tout le graphe d'objets connexe à l'objet passé en paramètre ou rendu en résultat est transmis, ***en respectant la structure du graphe***, et cela fonctionne ***même si le graphe a des boucles***.



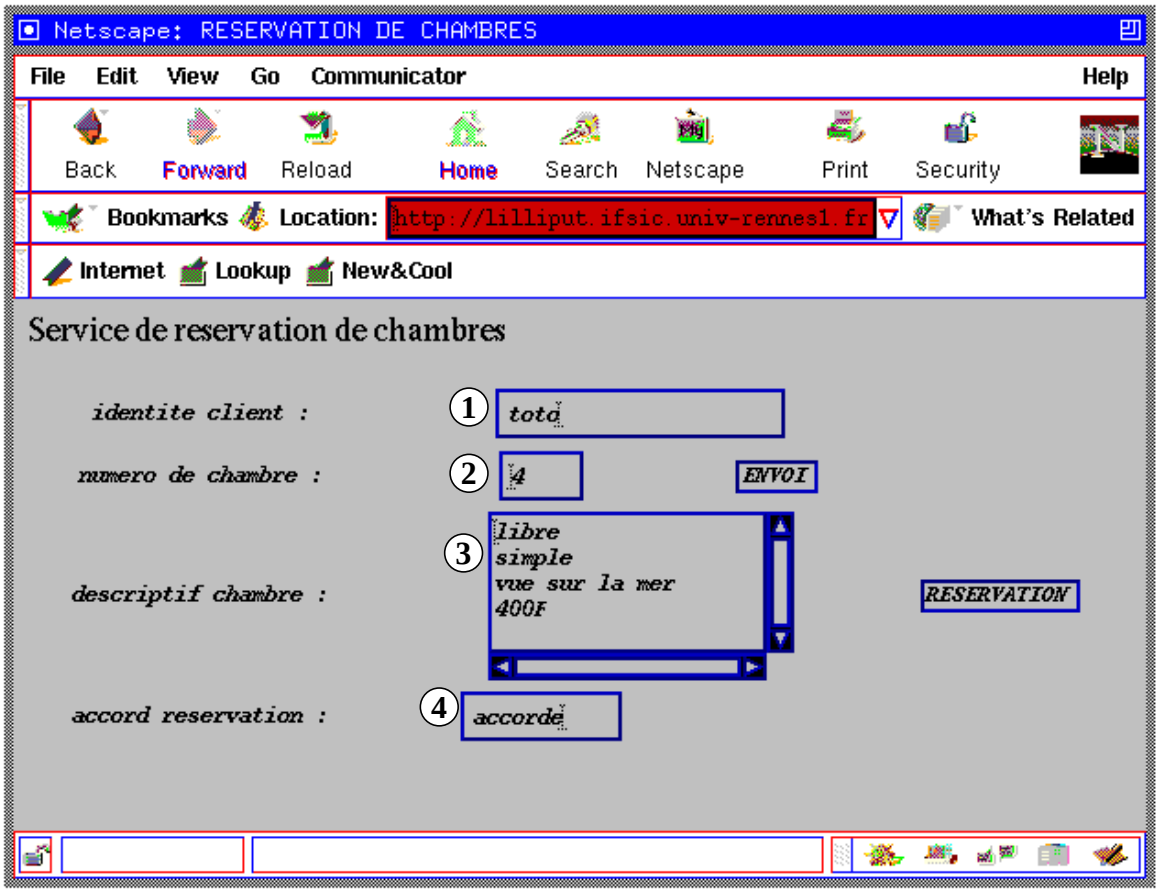
À titre d'exercice amusant, le lecteur pourra essayer de transmettre l'arbre généalogique de l'exercice 3 et vérifier que cela fonctionne.

© IFSIC - Université de Rennes 1 - 2003

Exercice 15

Réservation de chambres

On considère un système rudimentaire de réservation de chambres d'hôtel. La réservation se fait par internet, au moyen de la page Web illustrée sur la figure suivante.
Le client communique son identité au moyen de la zone de texte **(1)**. Il s'intéresse à une chambre en indiquant son numéro **(2)** et en cliquant sur le bouton ***ENVOI***. Le descriptif de la chambre apparaît dans la zone de texte **(3)**, indiquant si la chambre est libre ou non.
Si la chambre est libre et lui convient, le client peut tenter de la réserver en cliquant le bouton ***RESERVATION***. En réponse le système lui indique dans la zone de texte **(4)** si la réservation a pu être faite (mention ***accordé***) ou non (mention ***refusé***). Bien que le client ait pu voir la chambre libre, il est possible, à cause de la concurrence entre les clients, que la chambre soit réservée au moment de sa tentative de réservation.



Le squelette de la programmation Java de cette application est donné sur les pages suivantes. Cette programmation utilise la notion d'applet et les appels de méthodes à distance (`rmi`). Les zones marquées `----------` sont destinées à être complétées.

Les chambres de l'hôtel sont décrites par un tableau de chambres (10 chambres dans l'exemple), créé dans la classe **`ImplementServeurReservation`**.

© IFSIC - Université de Rennes 1 - 2005

```
public class ClientReservation extends Applet {

TextField identClient = new TextField(20);

TextField numChambre = new TextField(4);
Button boutonEnvoiChambre = new Button("ENVOI");
class ActionEnvoiChambre implements ActionListener {
  public synchronized void actionPerformed(ActionEvent e) {
    int numeroChambre;
    try {
     numeroChambre = Integer.parseInt(numChambre.getText());
     chambreCourante=serveurReservation.obtientChambre(numeroChambre);
     descriptif.setText(chambreCourante.descriptif());
   } catch(Exception ex){};
   accordReservation.setText("");
}}

TextArea descriptif = new TextArea(5,20);
Button boutonReservation = new Button("RESERVATION");
class ActionReservation implements ActionListener {
  public synchronized void actionPerformed(ActionEvent e) {
    if (chambreCourante == null) {return;}
    try { boolean ok =
           chambreCourante.tentativeReservation(identClient.getText());
     if (ok) {accordReservation.setText("accorde");}
     else {accordReservation.setText("refuse");}
   } catch(Exception ex){};
}}

TextField accordReservation = new TextField(10);

InterfaceReservation leServeur;
InterfaceChambre chambreCourante;

public void init(){
  Placement.p(this,new Label("identite client : "),1,1,1,1);
  Placement.p(this,identClient,2,1,6,1);
  Placement.p(this,new Label("numero de chambre : "),1,2,1,1);
  Placement.p(this,numChambre,2,2,1,1);
  Placement.p(this,boutonEnvoiChambre,3,2,1,1);
  Placement.p(this,new Label("descriptif chambre : "),1,3,1,1);
  Placement.p(this,descriptif,2,3,6,1);
  Placement.p(this,boutonReservation,8,3,1,1);
  Placement.p(this,new Label("accord reservation : "),1,4,1,1);
  Placement.p(this,accordReservation,2,4,1,1);

  try { leServeur = (InterfaceReservation) Naming.lookup(
         "//lilliput.ifsic.univ-rennes1.fr:8686/reservation");
  } catch(Exception e){System.out.println ("erreur lookup");}

  boutonEnvoiChambre.addActionListener(new ActionEnvoiChambre());
  boutonReservation.addActionListener(new ActionReservation());
}}
```


© IFSIC - Université de Rennes 1 - 2003

```
public interface InterfaceReservation extends Remote {
public InterfaceChambre obtientChambre(intnumChambre)
                                        throws RemoteException;
}
```

```
public interface InterfaChambre extends Remote {
public String descriptif() throws RemoteException;
public boolean tentativeReservation(String identClient)
                                                throws RemoteException;
}
```

```
public class Reservation ---------- {

public static Chambre[] chambre = new Chambre[10];

public Reservation() throws RemoteException {
 chambre[0]= new Chambre("simple", "vue sur la rue", 200);
 chambre[1]= new Chambre("double", "vue sur la rue", 300);
 chambre[2]= new ImplementChambre("simple", "vue sur la rue", 200);
 chambre[3]= new ImplementChambre("simple", "vue sur la mer", 400);
 chambre[4]= new ImplementChambre("simple", "vue sur la mer", 400);
 chambre[5]= new ImplementChambre("double", "vue sur la mer", 500);
 chambre[6]= new ImplementChambre("simple", "vue sur la rue", 200);
 chambre[7]= new ImplementChambre("simple", "vue sur la rue", 200);
 chambre[8]= new ImplementChambre("double", "vue sur la rue", 300);
 chambre[9]= new ImplementChambre("simple", "vue sur la mer", 350);
}



----------



public static void main(String[] arg) {
  try {
    // creation du serveur de reservation et enregistrement sur le reseau
    LocateRegistry.createRegistry(8686);
    Reservation leServeur = new Reservation();
    Naming.bind(
   "//lilliput.ifsic.univ-rennes1.fr:8686/reservation",leServeur);
    System.out.println("Serveur Reservation demarre");
  }
  catch(Exception e) {System.out.println("erreur"); return;}
}
}
```


© IFSIC - Université de Rennes 1 - 2005

```
public class Chambre  ---------- {
private final String capacite;
private final String vue;
private int tarif;
String etatReservation="libre";

public Chambre(String capacite, String vue, int tarif)
                                              throws RemoteException {
  this.capacite=capacite; this.vue=vue; this.tarif=tarif;
}


----------


----------



}
}
```


© IFSIC - Université de Rennes 1 - 2003

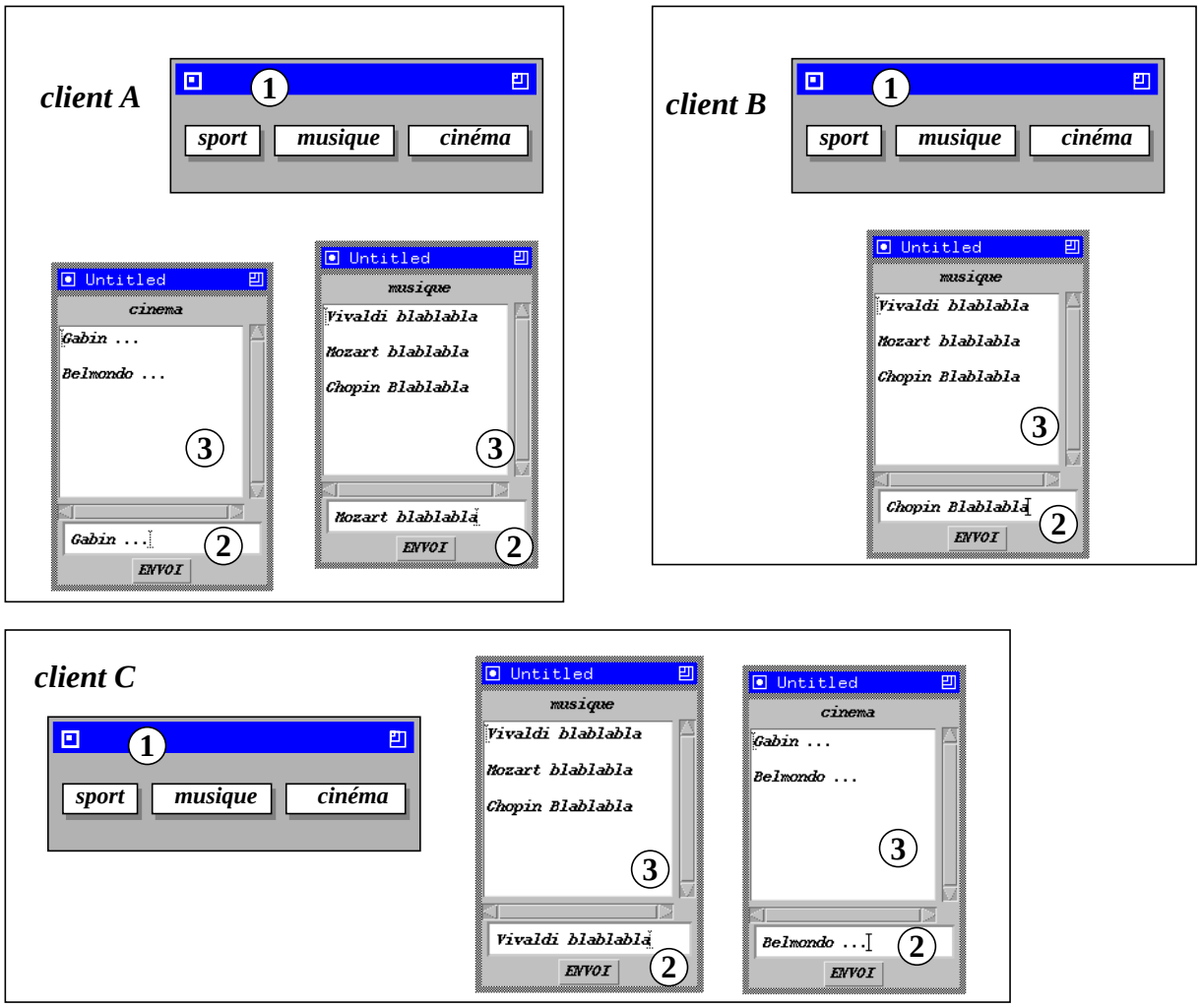
Exercice 16

Forum de discussion

On veut réaliser un forum permettant à des clients situés sur des machines diverses de participer à des discussions sur divers sujets, trois dans l'exemple : ***sport***, ***musique*** et ***cinéma***. Le client s'inscrit à un sujet en cliquant sur un des boutons de la fenêtre **(1)**, ce qui crée une fenêtre pour dialoguer sur ce sujet, composée de deux zones de texte, la zone **(2)** pour composer les messages et la zone **(3)** pour afficher tous les messages des divers protagonistes de la discussion. Un bouton ***ENVOI*** permet d'émettre le message nouvellement composé.

Le client peut se désinscrire d'un sujet en cliquant à nouveau sur le bouton du sujet.

Chaque client peut participer en même temps à autant de sujets de discussion qu'il veut. Dans l'exemple illustré sur la figure, les clients ***A*** et ***C*** sont inscrits aux sujets ***musique*** et ***cinéma*** et le client ***B*** est inscrit au sujet ***musique***. Un client reçoit uniquement les messages émis pendant qu'il est inscrit au sujet de discussion concerné (le serveur ne mémorise pas les messages, il ne fait que les diffuser).



© IFSIC - Université de Rennes 1 - 2005

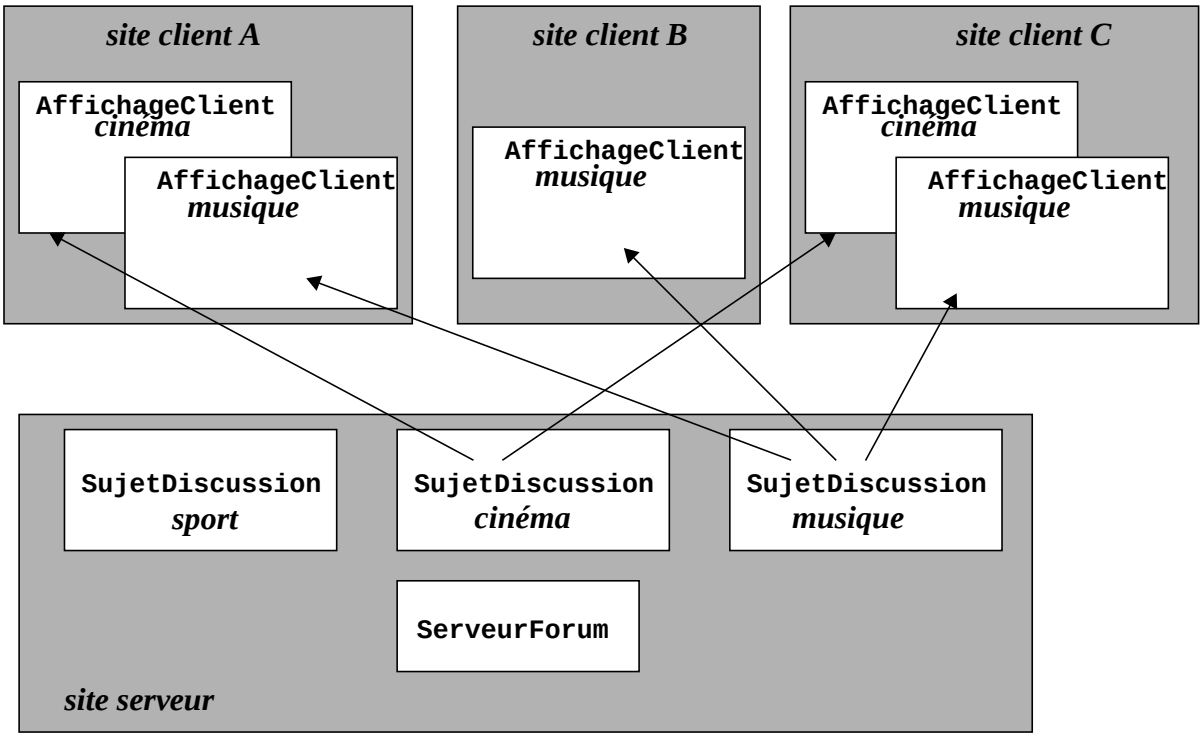
L'architecture de cette application, basée sur les appels de méthodes à distance (**`rmi`**), est illustrée sur le schéma suivant.

Sur le site serveur résident 2 types d'objets distants :

- Le serveur lui-même, défini par la classe **`ServeurForum`**, objet unique connu par les clients grâce à un nom externe de la forme
  "**`//machine:port/leServeur`**"
- Les sujets de discussion, définis par la classe **`SujetDiscussion`**.

Sur un site client réside un type d'objet distant, défini par la classe **`Affichage-Client`**. Il y a un objet de cette classe par sujet auquel est inscrit le client. Un tel objet est matérialisé par la fenêtre permettant la composition de message et l'affichage de la discussion.

Dans cette application simple, un sujet de discussion n'est rien de plus que la liste des affichages clients qui participent à la discussion. Il permet de diffuser le message émis par un client vers tous les autres participants.



Le programme de cette application est en grosse partie donné dans les pages suivantes. L'exercice consiste à comprendre les diverses pièces qui le constituent et à compléter cette programmation.

© IFSIC - Université de Rennes 1 - 2003

Interface d'objet distant pour le serveur de forum,
objet unique, localisé sur le site serveur,
connu par un nom externe "//machine:port/leServeur"

```
import java.rmi.*;

public interface InterfaceServeurForum extends Remote {

          public InterfaceSujetDiscussion obtientSujet(String titre)
                                                    throws RemoteException;
```

rend en résultat le sujet de discussion identifié par **`titre`** :
"sport", "musique", "cinema", ...
rend **`null`** si le titre ne correspond à aucun sujet

```
}
```

Interface d'objet distant pour les sujets de discussion : sport, musique, cinema, ...
objets localisés sur le site serveur
captés depuis les sites clients par références rendues en résultat par le site serveur

```
import java.rmi.*;

public interface InterfaceSujetDiscussion extends Remote {

public void inscription(InterfaceAffichageClient c) throws RemoteException;
```

inscrit l'afficheur client **`c`** à ce sujet de discussion

```
public void desInscription(InterfaceAffichageClient c)
                                                    throws RemoteException;
```

désinscrit **`c`**

```
public void diffuse(String Message) throws RemoteException;
```

diffuse le message à tous les afficheurs clients couramment inscrits à ce sujet

```
}
```

Interface d'objet distant pour les afficheurs clients d'un sujet de discussion
objets localisés sur les sites clients du forum, captés sur le site serveur par références passées en paramètre depuis les sites clients

```
import java.rmi.*;

public interface InterfaceAffichageClient extends Remote {

public void affiche(String Message) throws RemoteException;
```

affiche le message sur cet afficheur client

```
}
```


© IFSIC - Université de Rennes 1 - 2005

Implémentation de ServeurForum

```
import java.rmi.*; import java.rmi.server.*; import java.rmi.registry.*;

public class ServeurForum extends UnicastRemoteObject
                          implements InterfaceServeurForum {
SujetDiscussion sport;
SujetDiscussion musique;
SujetDiscussion cinema;

public ServeurForum() throws RemoteException {
  sport   = new SujetDiscussion("sport");
  musique = new SujetDiscussion("musique");
  cinema  = new SujetDiscussion("cinema")
}

public InterfaceSujetDiscussion obtientSujet(String titre)
                                 throws RemoteException {
  ...
}

public static void main(String[] arg) {
  try {
    // creation du serveur de forum et enregistrement sur le reseau
    LocateRegistry.createRegistry(8686);
    ServeurForum leServeur = new ServeurForum();
    Naming.bind(..., leServeur);
    System.out.println("demarrage du serveur");
  }
  catch(Exception e) {
   System.out.println("erreur enregistrement serveur"); return;
  }
}
}
```

Implémentation de SujetDiscussion

```
import java.rmi.*; import java.rmi.server.*; import list.*;

class SujetDiscussion  extends UnicastRemoteObject
                       implements InterfaceSujetDiscussion {
private final String titre;

// liste des protagonistes de ce sujet de discussion
private ...

public SujetDiscussion(String titre) throws RemoteException {
  this.titre=titre;
}

public synchronized void inscription(InterfaceAffichageClient c) { ... }

public synchronized void desInscription(InterfaceAffichageClient c){ ... }

public synchronized void diffuse(String message) { ... }
}
```


© IFSIC - Université de Rennes 1 - 2003

Implémentation de `AffichageClient`

```
import java.awt.*; import java.awt.event.*; import ihm.*;
import java.rmi.*; import java.rmi.server.*;


public class AffichageClient extends UnicastRemoteObject
                          implements InterfaceAffichageClient {

InterfaceSujetDiscussion sujetDiscussion;

Frame f = new Frame();

TextArea discussion = new TextArea(10,20);

TextField composeMessage = new TextField(20);

Button boutonEnvoi = new Button("ENVOI");
class ActionEnvoi implements ActionListener {
  public synchronized void actionPerformed(ActionEvent e) { ... }
}


public AffichageClient(String titre, InterfaceSujetDiscussion s)
                                              throws RemoteException {
  sujetDiscussion=s;
  Placement.p(f,new Label(titre),1,1,1,1);
  Placement.p(f,discussion,1,2,1,1);
  Placement.p(f,composeMessage,1,3,1,1);
  Placement.p(f,boutonEnvoi,1,4,1,1);
  boutonEnvoi.addActionListener(new ActionEnvoi());
  f.pack(); f.setVisible(true);
}


public void affiche(String Message) { ... }


public void termine() { f.dispose();}

}
```


© IFSIC - Université de Rennes 1 - 2005

ClientForum

```
import java.awt.*; import java.awt.event.*;
import ihm.*; import java.rmi.*; import java.rmi.registry.*;

public class ClientForum extends Frame {

Button boutonIncriptionSport   = new Button("sport");
Button boutonIncriptionMusique = new Button("musique");
Button boutonIncriptionCinema  = new Button("cinema");
class ActionInscription implements ActionListener {
  private String titre;
  private boolean inscrit=false;
  private AffichageClient c; // affichage client associe au bouton
  private InterfaceSujetDiscussion sujet; // sujet associe au bouton

  public ActionInscription(String titre) {
    ...
  }
  public synchronized void actionPerformed(ActionEvent e) {
    ...
  }
}


InterfaceServeurForum leServeur;

public ClientForum(){
  try {
    leServeur = (InterfaceServeurForum)
    Naming.lookup(...);
  } catch(Exception e){
    System.out.println ("erreur nommage serveur"); return;
  }

  Placement.p(this,boutonIncriptionSport,  1,1,1,1);
  Placement.p(this,boutonIncriptionMusique,2,1,1,1);
  Placement.p(this,boutonIncriptionCinema, 3,1,1,1);
  boutonIncriptionSport.addActionListener(...);
  boutonIncriptionMusique.addActionListener(...);
  boutonIncriptionCinema.addActionListener(...);
  setVisible(true); pack();
}



public static void main(String[] x){
  new ClientForum();
}
}
```


© IFSIC - Université de Rennes 1 - 2003

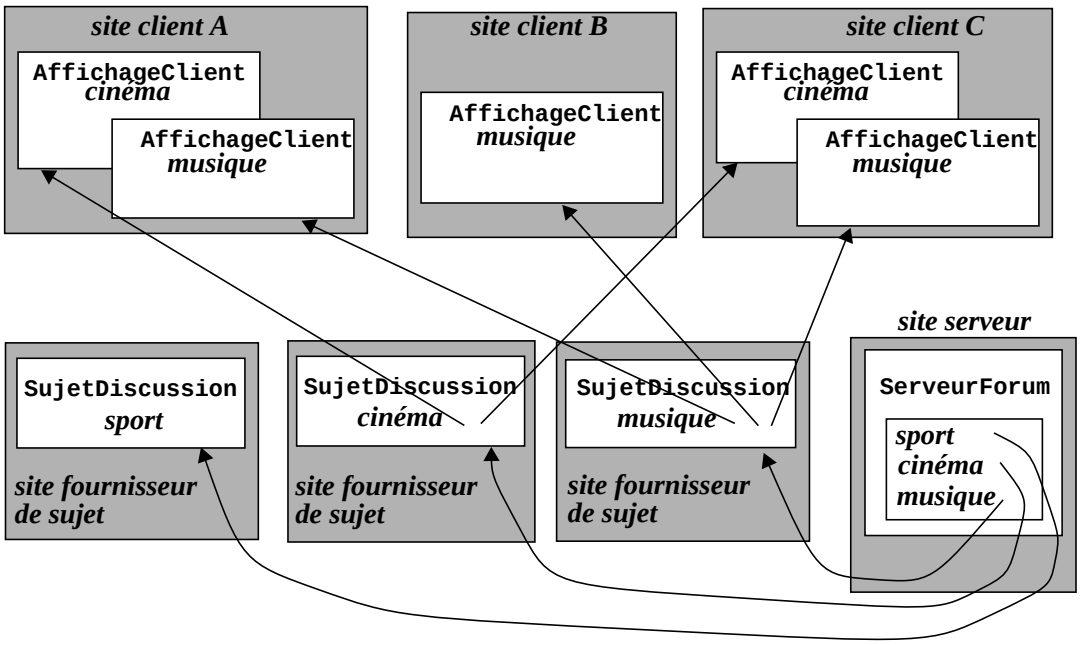
Exercice 17

Forum de discussion ammélioré

Nous voulons apporter les améliorations suivantes à l'application forum de discussion :

- ***Création dynamique de sujets de discussion*** : les sujets de discussions peuvent être créés à volonté.
- ***Répartition des sujets de discussion*** : pour éviter de surcharger le site du serveur du forum par les activités de diffusion des messages sur tous les sujets, on décide de répartir les sujets de discussion sur des sites distincts appelés ***sites fournisseurs***.

L'architecture de cette nouvelle version peut être illustrée par le schéma suivant :



Dans cet exemple, 3 sites fournisseurs de sujets se sont manifestés, pour les sujets respectifs *sport*, *cinéma* et *musique*. Le site serveur du forum se borne à conserver une table de correspondance entre les titres des sujets et les sujets situés sur les sites fournisseurs.

Un programme **`FournisseurDeSujet`**, exécuté sur un site fournisseur, crée un sujet de discussion et l'enregistre auprès du serveur de forum. Un fournisseur de sujet se lance par la commande : **`java FournisseurDeSujet`** ***titre***

Pour réaliser la table de correspondance entre titres et sujets, on utilisera la classe générique **`Table`** qui réalise une table d'associations <*clé*, *objet*> où les clés sont des chaînes de caractères de type **`String`** et les objets sont de type **`InterfaceSujetDeDiscussion`** :

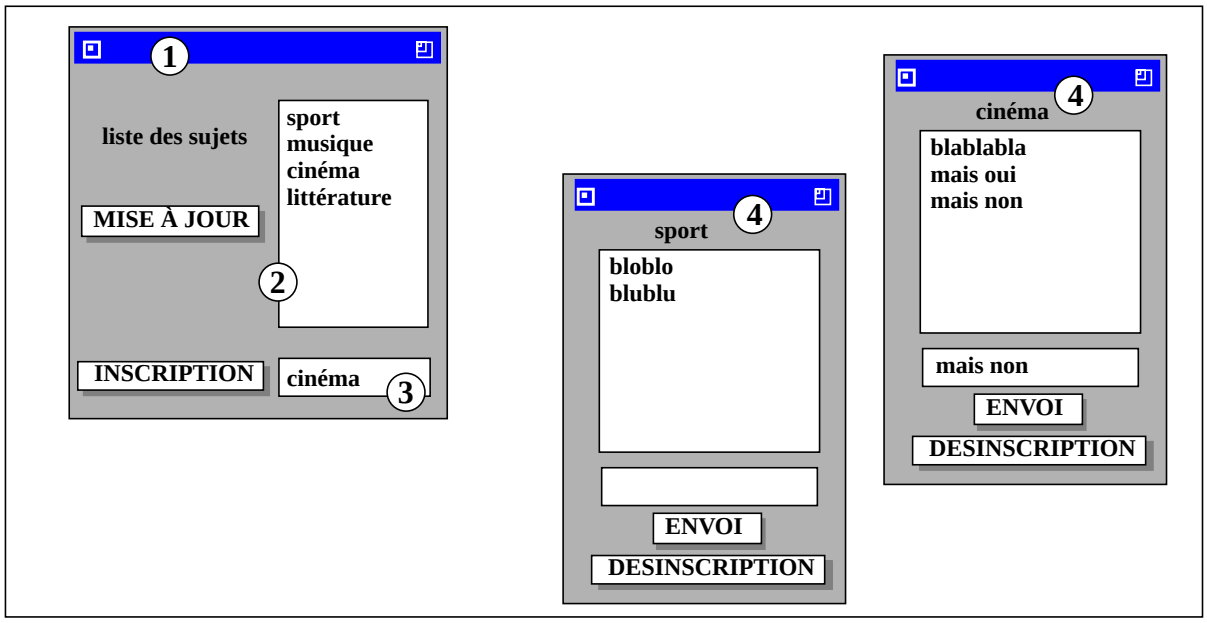
La table des sujets est initialement vide. Pour simplifier, on ne cherchera pas à supprimer un sujet existant et on suppose (sans le tester) que les fournisseurs de sujets proposent des titres tous différents.

© IFSIC - Université de Rennes 1 - 2005

À tout moment, un site client peut demander à obtenir la liste des sujets de façon à pouvoir s'inscrire à un ou plusieurs de ces sujets.

Les classes d'objets distants (accessibles à distance) sont toujours **`ServeurForum`**, **`SujetDiscussion`** et **`AffichageClient`**, mais les interfaces de certaines doivent être modifiées.
Déterminer quelles interfaces d'objets distants ***doivent*** être modifiées et rédiger ces (ou cette) interface(s). On cherchera à modifier le moins possible les interfaces de la version simple.

Rédiger l'application complète, avec l'interface homme-machine suivante pour les clients du forum.



La fenêtre **(1)** est la fenêtre initiale du client. La zone de texte **(2)** montre la liste des sujets disponibles. Un bouton **MISE À JOUR** permet de réactualiser cette liste.

Pour s'inscrire à un sujet, le client saisit le titre du sujet dans la zone de texte **(3)** et appuie sur le bouton **INSCRIPTION**. Lors d'une inscription, une nouvelle fenêtre **(4)**, de type **`AffichageClient`** apparaît. Pour des raisons de simplicité, on ne cherche pas à empêcher qu'un client s'inscrive plusieurs fois à un même sujet. D'ailleurs cela peut être intéressant pour lui d'avoir plusieurs fenêtres ouvertes sur le même sujet.

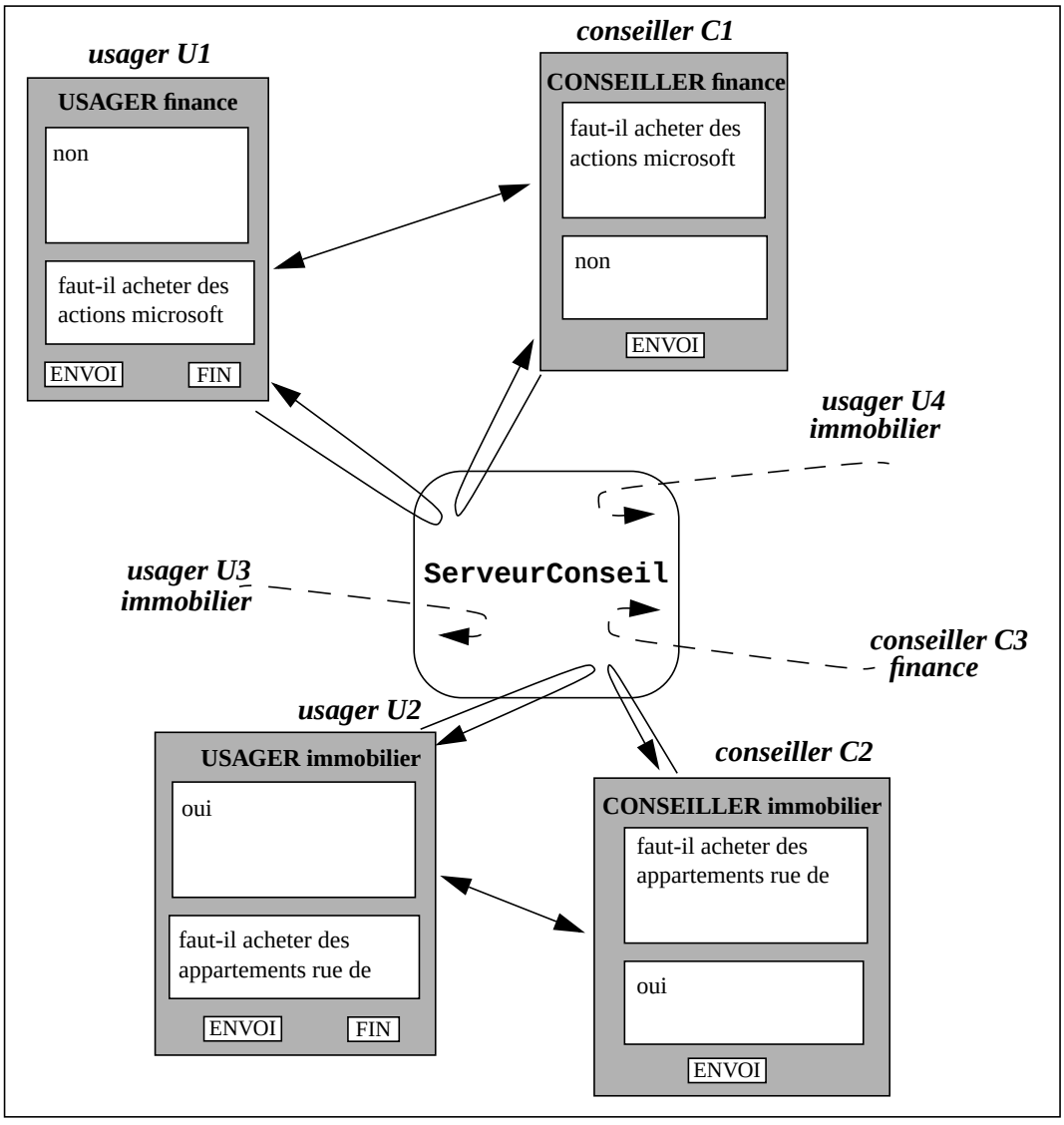
Remarque : les sujets étant ici en nombre dynamique, la désinscription d'un sujet est faite par un bouton associé à l'affichage de ce sujet.

© IFSIC - Université de Rennes 1 - 2003

Exercice 18

Agence de conseils

On se propose de réaliser une agence de conseils. Cette application permet de mettre en relation des ***usagers*** ayant besoin de conseils dans certains domaines (***finance***, ***immobilier***, ***informatique***...) et des ***conseillers*** spécialistes de ces domaines.



Le site serveur de l'agence exécute un programme **`ServeurConseil`** dont le rôle est de mettre en relation chaque usager avec un spécialiste du domaine demandé. Les usagers et les spécialistes ont leurs propres machines et peuvent être éloignés du site serveur. Il peut y avoir plusieurs conseillers pour un même domaine

Les conseillers, comme les usagers, sont a priori inconnus du serveur et se manifestent dynamiquement (en cours d'exécution du serveur).

Une fois la connexion établie entre eux, un usager et un conseiller communiquent directement, sans intervention du serveur, au moyen de fenêtres dotées de zones de textes pour saisir et afficher les questions et les réponses.

Les usagers disposent également d'un bouton **FIN** pour terminer une session de conseil.

Dans l'exemple illustré sur la figure, l'usager ***U1*** est en session avec le conseiller ***C1***, l'usager ***U2*** avec le conseiller ***C2***, les usagers ***U3*** et ***U4*** sont en attente de conseiller immobilier et le conseiller ***C3*** est en attente d'usager finance.

© IFSIC - Université de Rennes 1 - 2005

La programmation de cette application utilise les appels de méthodes à distance (**rmi**). Elle est constituée de 4 classes :

ServeurConseil
serveur de l'agence de conseil. Cette classe offre 2 méthodes :

```
InterfaceUsager proposeConseil
        (String titreDomaine, InterfaceConseiller c)
```

Appelée par le conseiller **c** pour proposer ses services de conseil pour le domaine intitulé **titreDomaine**. Cherche à connecter le conseiller **c** avec un usager ***u*** et rend cet usager ***u*** en résultat. Cette méthode est bloquante : tant qu'il n'y a pas d'usager demandeur pour le domaine, elle attend.

```
InterfaceConseiller demandeConseil
        (String titreDomaine, InterfaceUsager u)
```

Appelée par l'usager **u** pour obtenir la connexion avec un conseiller du domaine intitulé **titreDomaine**. Cherche à connecter l'usager **u** avec un conseiller ***c*** et rend ce conseiller ***c*** en résultat. Cette méthode est bloquante : tant qu'il n'y a pas de conseiller disponible pour le domaine, elle attend.

MoniteurAppariement
modèle de moniteur qui réalise les appariements entre usagers et conseillers pour un domaine. Cette classe offre 2 méthodes :

```
InterfaceUsager obtientUsager(InterfaceConseiller c)
```

Cherche à apparier le conseiller **c** avec un usager ***u*** et rend ***u*** en résultat, attend tant que ce n'est pas possible.

```
InterfaceConseiller obtientConseiller
                                (InterfaceUsager u)
```

Cherche à apparier l'usager **u** avec un conseiller ***c*** et rend ***c*** en résultat, attend tant que ce n'est pas possible.

Usager
classe qui réalise la logique d'un usager. Son programme principal **main** lit le domaine demandé puis crée une instance d'usager. Cette instance s'adresse au serveur pour obtenir la connexion avec un conseiller puis elle crée une fenêtre permettant le dialogue avec ce conseiller. Elle offre une méthode :

```
void afficheReponse(String laReponse)
```

Permet au conseiller connecté d'afficher les réponses aux questions.

Conseiller
classe qui réalise la logique d'un conseiller. Son programme principal **main** lit le domaine demandé puis crée une instance de conseiller. Cette instance s'adresse au serveur pour obtenir la connexion avec un usager puis elle crée une fenêtre permettant le dialogue avec cet usager. Elle offre 2 méthodes :

```
void poseQuestion(String laQuestion)
```

Permet à l'usager connecté de poser des questions.

```
void finConnexion()
```

Permet à l'usager de terminer la session.

© IFSIC - Université de Rennes 1 - 2003

Paquetages utilisés dans les exercices

Paquetage `es` : Entrées sorties clavier et fichiers textes

Entrées clavier : classe `Lecture`

```
package es;
import java.io.*;


public class Lecture {

  public static char unCar() {
  // effet : lit un caractère frappé
  // résultat : le caractère frappé
    char c;
    try { c = (char) System.in.read();}
    catch(IOException e) {c= (char) 0;};
    return c;
  }
```

```
  public static String chaine(String delimiteurs) {
  // prérequis : delimiteurs.length()!=0
  // effet : lit une séquence de caractères constituée de délimiteurs
  // puis de caractères autres que des délimiteurs.
  // les délimiteurs sont les caractères de délimiteurs.
  // résultat : la chaine formée des caractères compris entre les delimiteurs
    StringBuffer b = new  StringBuffer();
    char c=unCar();
    // ignore les delimiteurs de tete
    while (delimiteurs.indexOf(c)!=-1) {c=unCar();};
    // lit jusqu'au prochain delimiteur
    while (delimiteurs.indexOf(c)==-1) {b.append(c); c=unCar();};
    return b.toString();
  }


  public static String chaine() {
  // résultat : Lecture.chaine(" \n")
    return  chaine(" \n");
  }
```


© IFSIC - Université de Rennes 1 - 2005

```
  public static int unEntier() {
  // effet : lit une chaîne de caractères comprise entre les délimiteurs
  // ' ', '\r' ou '\n'
  // résultat : l'entier representé en decimal par cette chaîne
  // si la chaîne ne respecte pas la syntaxe d'un entier décimal,
  // affiche un message d'erreur et retourne 0.
    String s=Lecture.chaine(" \r\n");
    try { return Integer.parseInt(s);}
    catch(NumberFormatException e) {
      System.err.println("\nErreur lecture d'entier");
      System.err.println("valeur 0 retournée");
      return 0;
    }
  }


  public static float unReel() {
  // effet : lit une chaîne de caractères comprise entre les délimiteurs
  // ' ', '\r' ou '\n'
  // résultat : le flottant representé en format usuel par cette chaîne
  // si la chaîne ne respecte pas la syntaxe usuelle pour un nombre réel,
  // affiche un message d'erreur et retourne 0.
    String s=Lecture.chaine(" \r\n");
    try { return  (Float.valueOf(s)).floatValue();}
    catch(NumberFormatException e) {
      System.err.println("\nErreur lecture de reél");
      System.err.println("valeur 0 retournée");
      return 0;
    }
  }

}
```


© IFSIC - Université de Rennes 1 - 2003

Entrées fichiers textes : classe `LectureFichierTexte`

```
package es;
import java.io.*;

public class LectureFichierTexte {
// pour lire un fichier texte.
// Les délimiteurs d'entités lues sont par défaut
// l'espace et le passage à la ligne
// exemple d'utilisation :
// LectureFichierTexte monEntree = new LectureFichierTexte("monFichier.txt");
// String item1 = monEntree.lireChaine(); // lit une String
// int item2 = monEntree.lireUnEntier(); // lit un entier écrit en décimal
// monEntree.fermer();

  private BufferedReader leFichier;
  private char prochainCaractere;
  private boolean finDeFichier;
  private String nom;

  private void tenteDeLireProchainCaractere() {
    try {
      int x = leFichier.read();
      prochainCaractere = (char) x;
      if (x==-1) {finDeFichier = true;}
    }
    catch(IOException e){
      System.err.println("erreur de lecture du fichier "+nom);
      Thread.dumpStack();
    }
  }

  public LectureFichierTexte(String nom) {
  // initialise un accès en lecture sur le fichier de nom nom
  // erreur si le fichier n'existe pas
    this.nom=nom;
    try {
      leFichier = new BufferedReader(new FileReader(nom));
      finDeFichier = false;
      tenteDeLireProchainCaractere();
    }
    catch(FileNotFoundException e){
      System.err.println("fichier "+nom+" inexistant");
      Thread.dumpStack();
    }
  }

    public void fermer() { // fermeture du fichier
    try {leFichier.close();}
    catch(IOException e) {
      System.err.println("erreur lors de la fermeture du fichier "+nom);
      Thread.dumpStack();
   }
  }
```


© IFSIC - Université de Rennes 1 - 2005

```
  public char lireUnCar() { // lecture d'un caractère
    char courant = prochainCaractere;
    tenteDeLireProchainCaractere();
    return courant;
  }

  public String lireChaine(String delimiteurs) {
    // lecture d'une chaine comprise entre delimiteurs
    // ou jusqu'à fin de fichier.
    // rend la chaine vide si fin de fichier.
    if (finDeFichier()) {return "";}
    char c=lireUnCar();
    // ignore les delimiteurs de tete
    while (!finDeFichier() && delimiteurs.indexOf(c)!=-1) {c=lireUnCar();}
    if (finDeFichier()) {return "";}
    // lit jusqu'au prochain delimiteur ou fin de fichier
    StringBuffer b = new StringBuffer(); b.append(c);
    c=lireUnCar();
    while (!finDeFichier() && delimiteurs.indexOf(c)==-1) {
       b.append(c); c=lireUnCar();
    }
    if(delimiteurs.indexOf(c)==-1){b.append(c);}
    // consomme les éventuels delimiteurs suivants
    while (!finDeFichier() && delimiteurs.indexOf(prochainCaractere)!=-1) {
      c=lireUnCar();
    }
    return b.toString();
  }

  public String lireChaine() {
    // lecture d'une chaine comprise entre delimiteurs ' ', '\r' ou '\n'
    return lireChaine(" \r\n");
  }

  public int lireUnEntier() { // lecture d'un entier
    try { return Integer.parseInt(lireChaine());
    }
    catch(NumberFormatException e) {
      System.err.println("Erreur lecture d'entier sur fichier "+nom);
      System.err.println("valeur 0 retournée");
      return 0;
    }
  }

  public double lireUnReel() { // lecture d'un nombre réel
    try { return  (Double.valueOf(lireChaine())).floatValue();
    }
    catch(NumberFormatException e) {
      System.err.println("Erreur lecture de reél sur fichier "+nom);
      System.err.println("valeur 0 retournée");
      return 0;
    }
  }

    public boolean finDeFichier() { // indique si fin de fichier
    return finDeFichier;
  }

}
```


© IFSIC - Université de Rennes 1 - 2003

Sorties fichiers textes : classe `EcritureFichierTexte`

```
package es;
import java.io.*;

public class EcritureFichierTexte {
// pour ecrire dans un fichier texte
// exemple d'utilisation :
// EcritureFichierTexte maSortie = new EcritureFichierTexte("monFichier.txt");
// maSortie.ecrire("\nje m'appelle toto et j'ai "); // ecrit une String
// maSortie.ecrire(12); // ecrit un int (écriture décimale d'un entier)
// maSortie.ecrire(" ans.") // ecrit une String
// maSortie.fermer();

  private PrintWriter leFichier;
  private String nom;

  public EcritureFichierTexte(String nom) {
  // initialise un accès en écriture sur le fichier de nom nom
  // crée le fichier s'il n'existe pas
    this.nom=nom;
    try {
      leFichier = new PrintWriter(new FileOutputStream(nom));
    }
    catch(IOException e){
      System.out.println("erreur lors de la création du fichier "+nom);
      Thread.dumpStack();
    }
  }

  public void fermer() { // ferme le fichier
    leFichier.close();
  }

    public void ecrire(char c) { // écrit un caractère
    leFichier.print(c);
  }

  public void ecrire(String s) { // écrit une chaîne de caractères
    leFichier.print(s);
  }

  public void ecrire(int k) { // écrit un entier sous forme décimale
     leFichier.print(k);
  }

  public void ecrire(boolean b) { // écrit un booléen, "true" ou "false"
     leFichier.print(b);
  }

  public void ecrire(double x) { // écrit un réel sous forme usuelle
     leFichier.print(x);
  }
}
```


© IFSIC - Université de Rennes 1 - 2005

Paquetage `list` : Listes génériques

```
package parcours;

public interface Parcours<Element> {

  public void tete();
  // effet : positionne this en début de collection
  // ou en fin de parcours si la collection est vide

  public void suivant();
  // prérequis : this n'est pas en fin de parcours
  // effet : positionne this sur l'élément suivant

  public boolean estEnFin();
  // résultat : indique si this est en fin de parcours
  // (au delà du dernier)

  public Element elementCourant();
  // prérequis : this n'est pas en fin de parcours
  // résultat : l'élement courant désigné par this
}


public interface ParcoursBidirectionnel<Element>
                        extends Parcours<Element>{
  public void precedent();
  // prérequis : this n'est pas en fin de parcours
  // effet : positionne this sur l'élément precedent

  public void queue();
  // effet : positionne this sur l'élément de queue
  // ou en fin de parcours si la collection est vide
}



public interface ParcoursModificateur<TypeElement>
                        extends Parcours<TypeElement>{

  public void modifElement(TypeElement e);
  // prérequis : this n'est pas en fin de parcours
  // effet : modifie l'élément courant de this en lui affectant e

  public void ajouteElement(TypeElement e);
  // effet : insère e après l'élément courant de this, insère en
  // tête si this est en fin de parcours, et se positionne sur
  // l'élément ajouté

  public void retireElement();
  // prérequis : this n'est pas en fin de parcours
  // effet : retire l'élément courant de this et se positionne sur
  // le suivant de l'élément supprimé, ou en fin de parcours si
  // on a retiré le dernier

}
```


© IFSIC - Université de Rennes 1 - 2003

```
package list;

import parcours.*;

public class Liste<T> {

    private static class Maillon<TE> {
     Maillon<TE> suivant; Maillon<TE> precedent; TE element;
     Maillon(Maillon<TE> s,Maillon<TE> p,TE e) {
             suivant=s;precedent=p;element=e;
         }
    }

    private Maillon<T> tete;
    private Maillon<T> queue;

  //====================== parcoureur de liste ====================
    public class Parcours implements ParcoursBidirectionnel<T>,
                                     ParcoursModificateur<T>     {

     private Maillon<T> courant;

        public Parcours() {courant=tete;} // Parcours initialisé au debut

        private Liste<T> laListe() { return Liste.this; }

        public Parcours(Parcours p) {
        // Parcours initialisé avec l'état du Parcours p
          if (p.laListe()!=Liste.this) {
            System.out.println("erreur :\n"
                + "parcours initialisé avec celui d'une autre liste");
            Thread.dumpStack(); System.exit(0);
          }
          courant=p.courant;
     }


     public void tete() {
     // effet : positionne this en tete de liste
        // (parcours en fin si liste vide)
          courant=tete;
        }
     public void queue() {
     // effet : positionne this en queue de liste
        // (parcours en fin si liste vide)
          courant=queue;
        }
     public void suivant() {
     // effet : positionne this sur l'élément suivant,
     // ne fait rien si estEnFin()
          if (courant!=null) {courant=courant.suivant;}
        }

     public void precedent() {
     // effet : positionne this sur l'élément précédent,
     // ne fait rien si estEnFin()
          if (courant!=null) {courant=courant.precedent;}
     }
```


© IFSIC - Université de Rennes 1 - 2005

```
      public boolean estEnFin() {
   // résultat : indique si this est en fin
   // (au dela de la queue, en deça de la tete)
        return courant==null;
      }
      public T elementCourant() {
   // prérequis : this n'est pas en fin
   // résultat : l'élément courant de this
       return courant.element;
      }

      public void modifElement(T nouvelElement) {
   // prérequis : this n'est pas en fin
   // effet : remplace l'élément courant de this par nouvelElement
       if (courant!=null) {courant.element=nouvelElement;}
      }

      public void ajouteElement(T nouvelElement) {
   // effet : insère nouvelElement à la suite de l'élément courant de this
   // insère en tête si this est en fin de parcours
   // l'élément inséré devient l'élément courant
       if (courant!=null) {courant.element=nouvelElement;}
          Maillon<T> suivant; Maillon<T> nouveau;
       if (courant==null) { // chaine en tete
              suivant=tete;
          nouveau = new Maillon<T>(tete,null,nouvelElement);
              tete=nouveau; courant=nouveau;
          }
          else { // chaine sur courant
          suivant=courant.suivant;
          nouveau = new Maillon<T>(suivant,courant,nouvelElement);
              courant.suivant=nouveau; courant=nouveau;
          }
          if (suivant!=null) {suivant.precedent=courant;}
          else {// nouvel element de queue
              queue=courant;
          }
   }

   public void retireElement() {
   // prérequis : this n'est pas en fin
      // effet : retire l'élément courant de this
   // le suivant de l'élément retiré, s'il existe,
   // devient l'élement courant, sinon this passe en fin de parcours
        Maillon<T> suivant=courant.suivant;
        Maillon<T> precedent=courant.precedent;
     if (precedent!=null) {precedent.suivant=suivant;}
        else {tete=suivant;}
     if (suivant!=null) {suivant.precedent=precedent;}
        else {queue=precedent;}
        courant=suivant;
   }
}
//=====================================================================

public Liste() { // liste vide
    tete=null; queue=null;
}
```


© IFSIC - Université de Rennes 1 - 2003

```
public Liste<T>.Parcours nouveauParcours() {
// résultat : un nouveau parcours initialisé au début de this
    return new Parcours();
}

public Liste<T>.Parcours nouveauParcours
                      (Liste<T>.Parcours p) {
// résultat : nouveau parcours initialisé à l'état de parcours de p
    return new Parcours(p);
}

public boolean estVide() {return tete==null;}
// résulat : indique si this est vide

public String toString() {
// résultat : this en clair
    if (estVide()) {return "<>";}
    Parcours p= new Parcours();
    StringBuffer resul =
         new StringBuffer("< "+ p.elementCourant().toString());
    p.suivant();
    while (!p.estEnFin()) {
     resul.append(" | " + p.elementCourant().toString());
        p.suivant();
    }
    resul.append(" >"); return resul.toString();
}

public void ajouteEnQueue(T nouvelElement) {
// effet : ajoute nouvelElement en queue de this
     if (queue==null) { // cas liste vide
      queue = new Maillon<T>(null,null,nouvelElement);
         tete = queue;
     }
     else { // chaine en queue
         Maillon<T> exDernier = queue;
      queue = new Maillon<T>(null,exDernier,nouvelElement);
         exDernier.suivant=queue;

     }
}


public void ajouteEnTete(T nouvelElement) {
// effet : ajoute nouvelElement en tête de this
     if (tete==null) { // cas liste vide
      tete = new Maillon<T>(null,null,nouvelElement);
         queue = tete;
     }
     else { // chaine en tete
         Maillon<T> exPremier = tete;
      tete = new Maillon<T>(exPremier,null,nouvelElement);
         exPremier.precedent=tete;
     }
}
```


© IFSIC - Université de Rennes 1 - 2005

```
public T retireEnQueue() {
  // prérequis : this n'est pas vide
  // effet : retire l'élément de queue
  // résultat : l'élément retiré
      T resul = queue.element;
      queue = queue.precedent;
      if (queue!=null) {queue.suivant=null;}
      else {tete = null;}
      return resul;
  }

  public T retireEnTete() {
  // prérequis : this n'est pas vide
  // effet : retire l'élément de tête
  // résultat : l'élément retiré
      T resul = tete.element;
      tete = tete.suivant;
      if (tete!=null) {tete.precedent=null;}
      else {queue = null;}
      return resul;
  }


  public static <TE>  Liste<TE> aPartirDe(TE[] e){
  // résultat : une nouvelle liste contenant les éléments de e
      Liste<TE> resul=new Liste<TE>();
      for (int i=0;i<e.length;i++){
       resul.ajouteEnQueue(e[i]);
      }
      return resul;
  }


  public Liste(T[] e){ // liste contenant les éléments de e
      tete=null; queue=null;
      for (int i=0;i<e.length;i++){
       ajouteEnQueue(e[i]);
      }
  }

}
```


© IFSIC - Université de Rennes 1 - 2003

Paquetage `ihm` : Fenêtrage

```
package ihm;
import java.awt.*;

// procedures de placement de composants visuels dans un receptacle

public class Placement {

  static GridBagLayout placeur= new GridBagLayout();
  static GridBagConstraints c = new GridBagConstraints();

  // procedure generale de positionnement
  //-------------------------------------
  public static void p(
         Container cont, Component comp, int x, int y, int w, int h, int pos,
         int t, int l, int b, int r, double wx, double wy, int fill) {

     cont.setLayout(placeur);
     c.gridx=x; c.gridy=y; // position (en nbre de cellules) du coin nord-est
    c.gridwidth=w; c.gridheight=h; // largeur et hauteur (en nbre de cellules)
     c.fill=fill;  // directions d'expansion : NONE, BOTH, HORIZONTAL, VERTICAL
     c.anchor=pos; // position du composant dans ses cellules :
                   // CENTER, EAST,NORTHEAST, NORTH, NORTHWEST,
                   // WEST, SOUTHWEST, SOUTH, SOUTHEAST ...
     c.weightx=wx; c.weighty=wy; // ponderation de la distribution de l'espace
                                 // supplementaire en cas d'agrandissement
     c.insets = new Insets(t,l,b,r); // marges en pixels
     placeur.setConstraints(comp, c); cont.add(comp);
  };

  // placement d'un composant qui ne grossit pas
  //--------------------------------------------
  public static void p(Container cont, Component comp,
            int x, int y, int w, int h, int pos, int t, int l, int b, int r) {
     p(cont, comp, x, y, w, h, pos, t, l, b, r,
                            0.0, 0.0, GridBagConstraints.NONE);
  };

  // positionnement d'un composant sans marges qui ne grossit pas
  //-------------------------------------------------------------
  public static void p(Container cont, Component comp,
                    int x, int y, int w, int h, int pos) {
     p(cont, comp, x, y, w, h, pos,
                              0, 0, 0, 0, 1.0, 1.0,
                                                GridBagConstraints.NONE);
  };

  // positionnement au centre d'un composant sans marges qui ne grossit pas
  //------------------------------------------------------------------------
  public static void p(Container cont, Component comp,
                    int x, int y, int w, int h) {
     p(cont, comp, x, y, w, h,
                         GridBagConstraints.CENTER, 0, 0, 0, 0, 1.0, 1.0,
                                                  GridBagConstraints.NONE);
  };
}
```


© IFSIC - Université de Rennes 1 - 2005

Squelettes d'exercices

© IFSIC - Université de Rennes 1 - 2003

Squelette exercice 10 - Simulateur de circuits logiques

```
import java.awt.*; import java.awt.event.*;
import es.*; import list.*; import ihm.*;

class Bit {
public static char et(char b1,char b2) {
  switch (b1) {
     case '0' : return '0';
     case '1' : return b2;
     default  : if(b2=='0') return '0'; else return 'X';
  }
}

public static char non(char b) {
  switch (b) {
     case '0' : return '1';
     case '1' : return '0';
     default  : return 'X';
  }
}
}


class Fil {
...
}



abstract class CircuitDeBase {
//   Circuit + procedure d'evaluation topEval
// + liste de tous les circuits de base, membre statique
  ...
}


class ET extends CircuitDeBase {
  ...
}


class INV extends CircuitDeBase {
  ...
};



class NON_ET {
  ...
}


class VERROU { // verrou SetReset
  ...
}
```


© IFSIC - Université de Rennes 1 - 2005

```
class SimulVerrou {
private Fil ma0=new Fil();
private Fil ma1=new Fil();
private Fil s=new Fil();
private VERROU v=new VERROU(ma0,ma1,s);
private FournisseurDeBit ema0;
private FournisseurDeBit ema1;
private AfficheurDeBit affma0;
private AfficheurDeBit affma1;
private AfficheurDeBit affs;

public SimulVerrou(FournisseurDeBit pema0, FournisseurDeBit pema1,
        AfficheurDeBit paffma0,  AfficheurDeBit paffma1, AfficheurDeBit paffs){
   ema0=pema0; ema1=pema1; affma0=paffma0; affma1=paffma1; affs=paffs;
}

public void top() {
    ma0.ecrire(ema0.lire());
    ma1.ecrire(ema1.lire());
    Liste<CircuitDeBase>.Parcours pc = CircuitDeBase.tous.nouveauParcours();
    while (!pc.estEnFin()) {
       pc.elementCourant().topEval();
       pc.suivant();
    }
    Liste<Fil>.Parcours pf = Fil.tous.nouveauParcours();
    while (!pf.estEnFin()) {
       pf.elementCourant().actualise();
       pf.suivant();
    }
    affma0.ecrire(ma0.lire()); affma1.ecrire(ma1.lire());
    affs.ecrire(s.lire());
}
}

interface FournisseurDeBit {
public char lire();
}

interface AfficheurDeBit {
public void ecrire(char b);
}

class FenetreSimul extends Frame {

private Checkbox poussoirMa0 = new Checkbox("Mise a 0");
class Ma0 implements FournisseurDeBit {
    //...
}

private Checkbox poussoirMa1 = new Checkbox("Mise a 1");
class Ma1 implements FournisseurDeBit {
    //...
}

Button boutonTop = new Button("top");
class ActionTop implements ActionListener {
  public synchronized void actionPerformed(ActionEvent e) {simulateur.top();}
}
```


© IFSIC - Université de Rennes 1 - 2003

```
TextField affichageMa0 = new TextField(40);
class  AfficheMa0 implements AfficheurDeBit {
    //...
}

TextField affichageMa1 = new TextField(40);
class  AfficheMa1 implements AfficheurDeBit {
    //...
}

TextField affichageS = new TextField(40);
class  AfficheS implements AfficheurDeBit {
    //...
}

Button boutonQuit=    new Button("quit");
class ActionQuit implements ActionListener {
  public synchronized void actionPerformed(ActionEvent e) {System.exit(0);}
}

SimulVerrou simulateur;

public FenetreSimul() {  super("CONTROLE DE LA SIMULATION D'UN VERROU");

  Placement.p(this,poussoirMa0,1,1,1,1);
  Placement.p(this,poussoirMa1,1,2,1,1);
  Placement.p(this,boutonTop,  1,3,1,1);
  Placement.p(this,boutonQuit, 1,4,1,1);
  Placement.p(this,affichageMa0, 2,1,1,1);
  Placement.p(this,affichageMa1, 2,2,1,1);
  Placement.p(this,affichageS  , 2,3,1,1);
  boutonTop.addActionListener(new ActionTop());
  boutonQuit.addActionListener(new ActionQuit());
  pack(); setVisible(true);
  simulateur = new SimulVerrou(new Ma0(), new Ma1(),
                    new AfficheMa0(), new AfficheMa1(), new AfficheS());
}
}

public class Simul {
  static public void main(String[] argv) {new FenetreSimul();}
}
```


© IFSIC - Université de Rennes 1 - 2005

Squelette exercice 12 - Trieuse systolique

Version 1

```
import es.*;

class Tamp { // modele de tampon a une place
int tampon; boolean estVide=true;
synchronized void prod(int v) {
  if(!estVide){try {wait();} catch(InterruptedException e){};};
  tampon=v; estVide=false; notify();
}
synchronized int cons() {
  if(estVide){try {wait();} catch(InterruptedException e){};};
  int resul=tampon; estVide=true; notify(); return resul;
}}


//---------------------------------------------------------------------
class Trieuse {

  static final int infini = 99999999;
  static final int N = 5; // nombre d'emplacements


  class Emplacement extends Thread { // modele de processus emplacement

      //...

  }


  Tamp tamponE = new Tamp(); // tampon d'entree
  Tamp tamponS = new Tamp(); // tampon de sortie

  class Bouchon extends Tamp { // tampon de droite (tampon bidon)
      //...
  }
  Tamp bouchon= new Bouchon();


  public Trieuse() {
    // phase de construction de l'application :
    // cree N emplacements interconnectes par tampons

      //...

  // phase de fonctionnement
     System.out.println("entrer "+2*N+" valeurs entieres");
     for(int i=0; i<2*N; i++) { tamponE.prod(Lecture.unEntier());};
     System.out.println("valeurs triees");
     for(int i=0; i<2*N; i++) { System.out.print(tamponS.cons()+" ");};
     System.out.println("");
  }
}
//---------------------------------------------------------------------


public class TriPara {static public void main(String[] argv){new Trieuse();}}
```


© IFSIC - Université de Rennes 1 - 2003

Version 2

```
import es.*; import ihm.*; import java.awt.*;

//=========================================================================
class Tamp { // modele de tampon a une place

int tampon; boolean estVide=true;

synchronized void prod(int v) {
  if(!estVide){try {wait();} catch(InterruptedException e){};}
  tampon=v; estVide=false; notify();
}

synchronized int cons() {
  if(estVide){try {wait();} catch(InterruptedException e){};}
  int resul=tampon; estVide=true; notify(); return resul;
}
}
//=========================================================================


//=========================================================================
class Trieuse {

int N; // nombre d'emplacements
static final int infini = 999999;

// fenetre de visualisation de la trieuse
Frame fTrieuse = new Frame("TRIEUSE");
TextField txtResul= new TextField(25);

//---- modele d'afficheurs pour la paire min-max d'un emplacement -------
class BiAfficheur extends Panel {
TextField t1=new TextField(4); TextField t2=new TextField(4);
public BiAfficheur(String s1, String s2) {
  Placement.p(this,new Label(s2),1,1,1,1); Placement.p(this,t2,2,1,1,1);
  Placement.p(this,new Label(s1),1,2,1,1); Placement.p(this,t1,2,2,1,1);
  t1.setEditable(false); t1.setBackground(Color.white);
  t2.setEditable(false); t2.setBackground(Color.white);
}
public void afficher(int v1, int v2, int duree) {
  t1.setText(interpEnt(v1)); t2.setText(interpEnt(v2));
  try {Thread.sleep(duree); } catch(InterruptedException e){};
}
String interpEnt(int v) { // pour un affichage discret des infinis
  if (v==infini) return ""; else return String.valueOf(v);
}
}
//-------------------------------------------------------------------------


public void entrer(int v) {entreeTrieuse.prod(v);}

public int consulter(){
  int resul=sortieTrieuse.cons();
  txtResul.setText(txtResul.getText()+resul+" "); return resul;
}
```


© IFSIC - Université de Rennes 1 - 2005

```
//---------------- modele de processus emplacement ---------------------

class Emplacement extends Thread {
  BiAfficheur minmax = new BiAfficheur("min","max");

    //...

}
//-------------------------------------------------------------------------


Tamp entreeTrieuse = new Tamp(); // tampon d'entree de la trieuse
Tamp sortieTrieuse = new Tamp(); // tampon de sortie de la trieuse

class Bouchon extends Tamp {  // tampon de droite (tampon bidon)
    //...
}

Tamp bouchon= new Bouchon();


Trieuse(int nbEmplacements) { // construction de la trieuse
  N=nbEmplacements;
  Placement.p(fTrieuse,txtResul,0,3,1,1);

   // cree N emplacements interconnectes par tampons

      //...

  fTrieuse.pack(); fTrieuse.setVisible(true);
}
}
//=========================================================================



// programme principal (instanciation et utilisation d'une trieuse)
//=========================================================================
public class TriPara2 {
  static public void main(String[] argv){
  int N=5;
  Trieuse t=new Trieuse(N);
  System.out.println("entrer "+2*N+" valeurs entieres");
  for(int i=0; i<2*N; i++) { t.entrer(Lecture.unEntier());};
  for(int i=0; i<2*N; i++) { int x=t.consulter();}
  System.out.println("termine");
  Lecture.chaine(""); // attente que l'utilisateur acquitte le resultat
  System.exit(0);
}
}
//=========================================================================
```


© IFSIC - Université de Rennes 1 - 2003

Squelette exercice 14 - Applet "j'apprend à compter"

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


public class ApprendCompter extends Applet {

  static class StoppableThread extends Thread {
    private boolean stop;
    public StoppableThread(){stop=false;}
    public synchronized void ordreStop() {stop=true;}
    public synchronized boolean testeStop() {return stop;}
  }


  // variables d'état de l'automate et l'éventuelle réponse
  int etat;
  static final int poseQuestion    = 1;
  static final int reponseFournie  = 2;
  static final int delaiEcoule     = 3;
  int iRep;

  AudioClip nombre[]= new AudioClip[19];
  AudioClip plus; AudioClip non;
  AudioClip laReponse; AudioClip exact;
  TextField question = new TextField(10);

  Button boutonRep[] = new Button[19];
  class ActionRep implements ActionListener {
      //...
  }



  Questionneur q; // processus qui pose les questions

  class Questionneur extends StoppableThread {
    Random gen= new Random();
    public void run() {
      while (!testeStop()) {
        int i=gen.nextInt(10); int j=gen.nextInt(10);
        //...
      }
    }
  }




  Chrono c; // processus qui attend le delai accordé
            // à la réponse et réveille le questionneur
  class Chrono extends StoppableThread {
    //...
  }
```


© IFSIC - Université de Rennes 1 - 2005

```
  synchronized void attendre() {
  // procedure qui permet au questionneur d'attendre
  // soit une reponse, soit le delai ecoule
    try{wait();} catch(InterruptedException e){}
  }


  synchronized void reponse(int i) {
  // procedure appelee lorsque l'utilisateur fournit une reponse
  // reveille le questionneur dans l'etat reponseFournie avec iRep=i
    //...
  }


  synchronized void delaiEcoule() {
  // procedure appelee lorsque le delai est ecoule
  // reveille le questionneur dans l'etat delaiEcoule
    //...
  }


  static void prononce(AudioClip audio, int duree) {
  // prononce un morceau de son et attend une duree supposee suffisante
  // pour qu'il soit entierement prononce
    audio.play(); try {Thread.sleep(duree);} catch (InterruptedException e){}
  }


  public void init () {
  // organise le panneau d'affichage et initialise les elements de son
    //...
  }


  public void start () { // demarre un questionneur
    //...
  }


  public void stop () { // arrete le questionneur
    //...
  }

}
```


© IFSIC - Université de Rennes 1 - 2003