Programmation objet en Java

M. Benameur / M. Tondeur

Concepts de base de la programmation Java

Nasser Benameur / Tondeur Hervé

Caractéristiques du langage

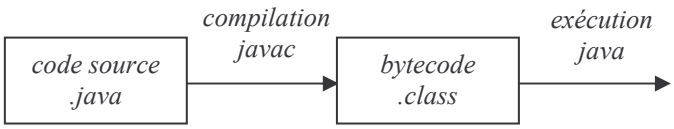
Java est un langage orienté objet, il à une syntaxe proche du C++, le compilateur est fourni avec les packages et les API nécessaire.

Java est un langage interprété, il est de ce fait portable, il est multithreaded, il n'utilise pas de pointeurs, pas de structures, pas de fonction globales.

La machine virtuelle Java (*JVM*)

Java est un langage *multi-plates-formes* qui permet, selon *Sun Microsystems*, son concepteur, d'écrire une fois pour toute des applications capables de fonctionner dans tous les environnements. Cet objectif est atteint grâce à l'utilisation d'une machine virtuelle Java (JVM) qui exécute les programmes écrits dans ce langage.

Compilation



- compilation du code source : *javac *.java* ;
- exécution sur la JVM : *java MainFile*.

Les classes

Définition

La classe regroupe la définition des *membres de classe*, c'est-à-dire :
- des *méthodes*, les opérations que l'on peut effectuer ;
- des *champs*, les variables que l'on peut traiter ;
- des *constructeurs*, qui permettent de créer des objets ;
- et encore d'autres choses plus particulières.

Plus précisément, une classe peut contenir des *variables (primitives ou objets)*, des *classes internes*, des *méthodes*, des *constructeurs*, et des *finaliseurs*.

La déclaration d'une classe se fait de la façon suivante :

```
[Modificateurs] class NomClasse

        {
              corps de la classe
        }
```

Le nom de la classe doit débuter par une majuscule.

Considérons l'exemple suivant :

```
Class Animal {
```

```
    // champs

    boolean vivant ;
    int âge ;

    // constructeurs

    Animal() {
    }

    // méthodes

    void viellit() {
          ++âge ;
    }

    void crie() {
    }

}
```

Les classes *final*

Une classe peut être déclarée *final*, dans un but de sécurité ou d'optimisation. Une classe *final* ne peut être étendue pour créer des sous-classes. Par conséquent, ses méthodes ne peuvent pas être redéfinies et leur accès peut donc se faire sans recherche dynamique.

Les classes internes

Une classe Java peut contenir, outre des primitives, des objets (du moins leurs références) et des définitions de méthodes, des définitions de classe. Nous allons maintenant nous intéresser de plus près à cette possibilité.

- **Plusieurs classes dans un même fichier**

Il arrive fréquemment que certaines classes ne soient utilisées que par une seule autre classe. Considérons l'exemple suivant :

```
Class Animal {

    // champs

    boolean vivant ;
    int âge ;
    Coordonnées position ;

    // constructeurs

    Animal() {
          position = new Coordonnées() ;
          }

    ...

    }

Class Coordonnées {

    // champs
```

```
int x = 0;
int y = 0;

...

}
```

Lors de la compilation du précédent fichier *Animal.java*, le compilateur produit deux fichiers : *Animal.class* et *Coordonnées.class*.

- **Les classes imbriquées ou *static***

Il peut être avantageux dans certains cas de placer la définition d'une classe à l'intérieur d'une autre, lorsque celle-ci concerne uniquement « la classe principale ». Voyons pour notre exemple :

```
Class Animal {

   // champs

   boolean vivant ;
   int âge ;
   Coordonnées position ;

   // classes imbriquées

   static Class Coordonnées {

         // champs

         int x = 0;
         int y = 0;

         ...

      }

   // constructeurs

   Animal() {
         position = new Coordonnées() ;
         }

   ...

   }
```

La définition de la classe *Coordonnées* est maintenant imbriquée dans la classe *Animal*. Par ailleurs, la référence à la classe *Coordonnées* devient *Animal.Coordonnées*. De manière générale, les références à une classe imbriquée en Java se font en utilisant le point comme séparateur.

Lors de la compilation du fichier ci-dessus, *Animal.java*, le compilateur produit deux fichiers : *Animal.class* et *Animal$Coordonnées.class* pour la classe imbriquée.

Quant au chemin d'accès, notons que *mesclasses.Animal* désigne la classe *Animal* dans le package *mesclasse,* tandis que *mesclasses.Animal.Coordonnées* désigne la classe *Coordonnées* imbriquée dans la classe *Animal,* elle-même contenu dans le package *mesclassses*. Ainsi il est possible d'utiliser la directive *import* pour importer les classes imbriquées explicitement ou en bloc :

```
import mesclasses.Animal.Coordonnées ;
import mesclasses.Animal.* ;
```

Les classes imbriquées peuvent elles-mêmes contenir d'autres classes imbriquées, sans limitation de profondeur, du moins du point de vue de Java.

Un dernier point. La classe *Coordonnées* a été déclarée *static*, ce qui est obligatoire pour toute classe imbriquée. En revanche, les interfaces imbriquées sont automatiquement déclarées *static* et il n'est donc pas nécessaire de les déclarer explicitement comme telles.

Les champs

Définition

L'état représente l'ensemble des variables qui caractérisent une classe ; on parle encore de champs ou de membres. Notons que Java initialise par défaut les variables membres.

Considérons l'exemple suivant :

```
Class Animal {

   // champs

   int âge ;
   static int longévité = 100 ;

}
```

Variables d'instances & Variables *static*

Dans l'exemple ci-dessus, *âge* est une variable d'instance, tandis que *longévité* représente une variable *static*.

Dans cet exemple, nous avons considéré que la *longévité* était une caractéristique commune à tous les animaux, mettons 100 ans ! Il n'est donc pas nécessaire de dupliquer cette information dans chacune des instances de la classe. Nous avons donc choisi de déclarer *longévité* comme une variable *static*. Il en résulte que cette variable appartient à la classe et non à ses instances.

Pour comprendre cette nuance, considérons une instance de la classe *Animal,* appelé *monAnimal*. L'objet *monAnimal* possède sa propre variable *âge*, à laquelle il est possible d'accéder grâce à la syntaxe :

```
monAnimal.âge
```

L'objet *monAnimal* ne possède pas de variable *longévité*. Normalement, *longévité* appartient à la classe *Animal*, et il est possible d'y accéder en utilisant la syntaxe :

```
Animal.longévité
```

Cependant, Java nous permet également d'utiliser la syntaxe :

```
monAnimal.longévité
```

Mais il faut bien comprendre que ces deux expressions font référence à la même variable. On peut utiliser le nom de la variable seul pour y faire référence, uniquement dans la définition de la classe.

Les variables *final*

Une variable déclarée *final* ne peut plus voir sa valeur modifiée. Elle remplit alors le rôle de constante dans d'autres langages. Une variable *final* est le plus souvent utilisée pour encoder des valeurs constantes.

Par exemple, on peut définir la constante *Pi* de la manière suivante :

```
final float pi = 3.14 ;
```

Déclarer une variable *final* offre deux avantages. Le premier concerne la sécurité. En effet, le compilateur refusera toute affectation ultérieure d'une valeur à la variable. Le deuxième avantage concerne l'optimisation du programme. Sachant que la valeur en question ne sera jamais modifié, le compilateur est à même de produire un code plus efficace. En outre, certains calculs préliminaires peuvent être effectués.

Les méthodes

Les méthodes sont les opérations ou les fonctions que l'on peut effectuer sur une classe. On distingue deux types de méthodes :

- les *accesseurs*, qui ne modifient pas l'état et se contente de retourner la valeur d'un champs ;
- les *modificateurs*, qui modifient l'état en effectuant un calcul spécifique.

Une déclaration de méthode est de la forme suivante :

```
[Modificateurs] Type nomMéthode ( paramètres ...)

      {
            corps de la méthode
      }
```

Le nom de la méthode débute par une minuscule ; la coutume veut qu'un accesseur débute par le mot « *get* » et qu'un modificateur débute par le mot « *set* ».

Les méthodes *static*

Les méthodes peuvent également être déclaré *static*. Imaginons que nous souhaitons construire un *accesseur* pour la variable *longévité* (voir exemple précédent). Nous pouvons le faire de la façon suivante :

```
Class Animal {

   // champs

   int âge ;
   static int longévité = 100 ;

   // méthodes

   static int getLongévité() {
         return longévité ;
   }

}
```

La méthode *getLongévité* peut être déclaré *static* car elle ne fait référence qu'à des membres *static* (en l'occurrence, la variable *longévité*). Ce n'est pas une obligation. Le programme fonctionne aussi si la méthode n'est pas déclaré *static*. Dans ce cas, cependant, la méthode est dupliquée chaque fois qu'une instance est créée, ce qui n'est pas très efficace.

Comme dans le cas des variables, les méthodes *static* peuvent être référencées à l'aide du nom de la classe ou du nom de l'instance. On peut utiliser le nom de la méthode seul , uniquement dans la définition de la classe.

Il est important de noter que les méthodes *static* ne peuvent en aucun cas faire référence aux méthodes ou aux variables non *static* de la classe. Elles ne peuvent non plus faire référence à une instance. (La référence *this* ne peut pas être employé dans la méthode *static.*)

Les méthodes *static* ne peuvent pas non plus être redéfinies dans les classes dérivées.

Les méthodes *final*

Les méthodes peuvent également être déclaré *final*, ce qui restreint leur accès d'une toute autre façon. En effet, les méthodes *final* ne peuvent pas être redéfinies dans les classes dérivées. Ce mot clé est utilisé pour s'assurer que la méthode d'instance aura bien le fonctionnement déterminé dans la classe parente. (S'il s'agit d'une méthode *static*, il n'est pas nécessaire de la déclarer *final* car les méthodes *static* ne peuvent jamais être redéfinies.)

Les méthodes *final* permettent également au compilateur d'effectuer certaines optimisations qui accélèrent l'exécution du code. Pour déclarer une méthode *final*, il suffit de placer ce mot clé dans sa déclaration de la façon suivante :

```
final int calcul(int i, int j) {...}
```

Le fait que la méthode soit déclaré *final* n'a rien à voir avec le fait que ces arguments le soient ou non.

Nous reviendrons sur l'utilité des méthodes *final* dans le chapitre concernant le *polymorphisme*, et notamment le concept *early & late binding*.

Les constructeurs

Les constructeurs : création d'objets

Les constructeurs et les initialiseurs sont des éléments très importants car ils déterminent la façon dont les objets Java commencent leur existence. Ces mécanismes, servant à contrôler la création d'instance de classe, sont fondamentalement différents des méthodes.

- **Les constructeurs (*constructor*)**

Les constructeurs sont des méthodes particulières en ce qu'elles portent le même nom que la classe à laquelle elles appartiennent. Elles sont automatiquement exécutées lors de la création d'un objet. Le constructeur par défaut ne possèdent pas d'arguments.

- Les constructeurs n'ont pas de type et ne retournent pas.
- Les constructeurs ne sont pas hérités par les classes dérivées.
- Lorsqu'un constructeur est exécuté, les constructeurs des classes parentes le sont également. C'est *le chaînage des constructeurs*. Plus précisément, si en première instruction le compilateur ne trouvent pas un appel à *this(...)* ou *super(...)*, il rajoute un appel à *super(...)*. L'utilisation de *this(...)* permet de partager du code entre les constructeurs d'une même classe, dont l'un au moins devra faire référence au constructeur de la super-classe.
- Une méthode peut porter le même nom qu'un constructeur, ce qui est toutefois formellement déconseillé.

- **Exemple de constructeurs**

Considérons l'exemple suivant :

```
class Animal {

   // champs

   boolean vivant ;
   int âge ;

   // constructeurs
```

```
    Animal() {
    }

    Animal(int a) {
          âge = a ;
          vivant = true ;
    }

    // méthodes

}
```

Si nous avions donné au paramètre *a* le même nom que celui du champs *âge*, il aurait fallu accéder à celle-ci de la façon suivante :

```
    Animal(int âge) {
          this.âge = âge ;
          vivant = true ;
    }
```

Toutefois, pour plus de clarté, il vaut mieux leur donner des noms différents. Dans le cas de l'initialisation d'une variable d'instance à l'aide d'un paramètre, on utilise souvent pour le nom du paramètre la première (ou les premières) lettre(s) du nom de la variable d'instance.

- **Création d'objets (*object*)**

Tout objet *java* est une *instance* d'une classe. Pour allouer la mémoire nécessaire à cet objet, on utilise l'opérateur *new*, qui lance l'exécution du constructeur.

La création d'un *Animal* se fait à l'aide de l'instruction suivante :

```
    Animal nouvelAnimal = new Animal(3) ;
```

- **Surcharger les constructeurs**

Les constructeurs, tout comme les méthodes, peuvent être surchargés dans le sens où il peut y avoir plusieurs constructeurs dans une même classe, qui possèdent le même nom (celui de la classe). Un constructeur s'identifie de part sa signature qui doit être différente d'avec tous les autres constructeurs.

Supposons que la plupart des instances soient créées avec 0 pour valeur initiale de *âge*. Nous pouvons alors réécrire la classe *Animal* de la façon suivante :

```
class Animal {

    // champs

    boolean vivant ;
    int âge ;

    // constructeurs

    Animal() {
          âge = 0 ;
          vivant = true ;
    }

    Animal(int a) {
          âge = a ;
          vivant = true ;
    }
```

```
    // méthodes

}
```

Ici, les deux constructeurs possèdent des signatures différentes. Le constructeur sans paramètre traite le cas où l'âge vaut 0 à la création de l'instance. Une nouvelle instance peut donc être créée sans indiquer l'âge de la façon suivante :

```
Animal nouvelAnimal = new Animal() ;
```

- **Autorisation d'accès aux constructeurs**

Les constructeurs peuvent également être affectés d'une autorisation d'accès. Un usage fréquent de cette possibilité consiste comme pour les variables, à contrôler leur utilisation, par exemple pour soumettre l'instanciation à certaines conditions.

Initialisation des objets

Il existe en Java trois éléments pouvant servir à l'initialisation :
- *les constructeurs*,
- *les initialiseurs de variables d'instances et statiques*,
- *les initialiseurs d'instances et statiques*.

Nous avons déjà présenté l'initialisation utilisant un constructeur. Voyons les deux autres manières.

- **Les initialiseurs de variables d'instances et statiques**

Considérons la déclaration de variable suivante :

```
int a ;
```

Si cette déclaration se trouve dans une méthode, la variable n'a pas de valeurs. Toute tentative d'y faire référence produit une erreur de compilation.

En revanche, s'il s'agit d'une *variable d'instance* (dont la déclaration se trouve en dehors de toute méthode), Java l'initialise automatiquement au moment de l'instanciation avec une valeur par défaut. Pour *les variables statiques*, l'initialisation est réalisée une fois pour toute à la première utilisation de la classe.

Les variables de type numérique sont initialisées à *0*. Le type booléen est initialisé à *false*.

Nous pouvons cependant initialiser nous-mêmes les variables de la façon suivante :

```
int a = 1 ;
int b = a*7 ;
float c = (b-c)/3 ;
boolean d = (a < b) ;
```

Les initialiseurs de variables permettent d'effectuer des opérations d'une certaine complexité, mais celle-ci est tout de même limitée. En effet, ils doivent tenir sur une seule ligne. Pour effectuer des opérations plus complexes, il convient d'utiliser les constructeurs ou encore les initialiseurs d'instances.

- **Les initialiseurs d'instances**

Un initialiseur d'instance est tout simplement placé, comme les variables d'instances, à l'extérieur de toute méthode ou constructeur.

Voyons l'exemple suivant :

```
class Exemple {
```

```
    // champs

    int a ;
    int b ;
    float c ;
    boolean d ;

    // initialiseurs

    {
          a = 1 ;
          b = a*7 ;
          c = (b-a)/3 ;
          d = (a < b);
    }

}
```

- **Les initialiseurs statiques**

Un *initialiseur statique* est semblable à un *initialiseur d'instance*, mais il est précédé du mot *static*. Considérons l'exemple suivant :

```
class Voiture {

    // champs

    static int capacité ;

    // initialiseurs

    static {
          capacité = 80;
          System.out.println("La variable vient d'être initialisée.\n") ;
    }

    // constructeurs

    Voiture() {
    }

    // méthodes

    static int getCapacité() {
          return capacité;
          }

}
```

L'initialiseur statique est exécuté au premier chargement de la classe, que ce soit pour utiliser un membre statique, `Voiture.getCapacité()` ou pour l'instancier, `Voiture maVoiture = new Voiture()`.

Les membres statiques (ici la variable `capacité`) doivent être déclarés avant l'initialiseur. Il est possible de placer plusieurs initialiseurs statiques, où l'on souhaite dans la classe. Ils seront tous exécutés au premier chargement de celle-ci, dans l'ordre où ils apparaissent.

La destruction des objets (*garbage collector*)

Avec certains langages, le programmeur doit s'occuper lui-même de libérer la mémoire en supprimant les objets devenus inutiles. Avec Java, le problème est résolu de façon très simple : un programme, appelé *garbage*

collector, ce qui signifie littéralement « ramasseur d'ordures », est exécuté automatiquement dès que la mémoire disponible devient inférieure à un certain seuil. De cette façon, aucun objet inutilisé n'encombrera la mémoire.

Le concept de l'héritage

Hiérarchie des classes

De plus chaque classe *dérive* d'une classe de niveau supérieur, appelée *sur-classe*. Cela est vrai pour toutes les classes sauf une. Il s'agit de la classe *Object*, qui est l'ancêtre de toutes les classes.

Toute instance d'une classe est un objet du type correspondant, mais aussi du type de toutes ses classes ancêtres.

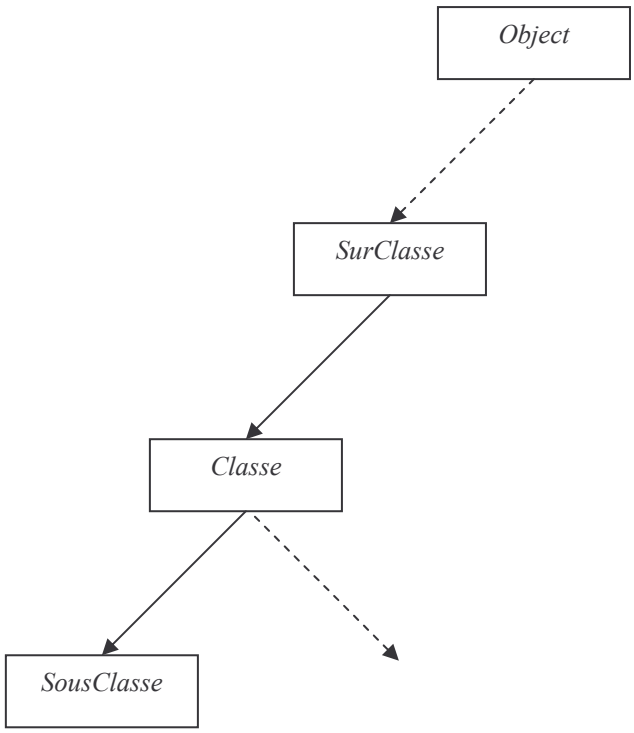
SousClasse hérite de toutes les caractéristiques de *Classe*, et donc, par transitivité, de *SurClasse*.

Une *classe* est toujours construite à partir d'une autre classe dont elle est *dérivée*. Une classe dérivée est une *sous-classe* d'une *sur-classe*.

- **Extends**

Lorsque le paramètre *extends* est omis, la classe déclarée est une sous classe de l'objet *Objet*.

- **Référence à la classe parente**



Appel du constructeur de la SurClasse

Il faut utiliser le mot réservé super()

Exemple :

```
Class Point
{int x,y ;
Point(int x, int y) {…}
Void affiche(){…}
```

```
}

class Point3D extends Point
{int z ;
Point3D(int a,int b,int c)
{super(a,b) ;
z=c ;}

}
```

Redéfinition des champs et des méthodes

- **Redéfinition des méthodes**

Une deuxième déclaration d'une méthode dans une classe dérivée remplace la première. Voyons un exemple avec la méthode *crie( )* redéfinie dans la classe *Chien* dérivée de la classe *Animal*.

```
Class Animal {

   // champs

   // méthodes

   void crie() {
   }

}

Class Chien extends Animal {

   // champs

   // méthodes

   void crie() {
         System.out.println("Ouah-Ouah !") ;
   }

}
```

La surchage

- **Surcharger les méthodes**

Une méthode est dite surchargée si elle permet plusieurs passages de paramètres différents.

Accessibilité

En Java, il existe quatre catégories d'autorisations d'accès, spécifiés par les modificateurs suivants : *private, protected, public*. La quatrième catégorie correspond à l'absence de modificateur.

Nous allons les présenter en partant du moins restrictif jusqu'au plus restrictif.

- **public**

Les classes, les interfaces, les variables (primitives ou objets) et les méthodes peuvent être déclarées *public*.

Les éléments *public* peuvent être utilisés par n'importe qui sans restriction ; il est accessible à l'extérieur de la classe.

- **protected**

Cette autorisation s'applique uniquement aux membres de classes, c'est-à-dire aux variables (objets ou primitives), aux méthodes et aux classes internes. Les classes qui ne sont pas membre d'une autre classe ne peuvent pas être déclarées *protected*.

Dans ce cas, l'accès en est réservé aux méthodes des classes appartenant au même *package*, aux classes dérivées de ces classes, ainsi qu'aux classes appartenant aux mêmes *packages* que les classes dérivées. Plus simplement, on retiendra qu'un élément déclaré *protected* n'est visible que dans la classe où il est défini et dans ses sous-classes.

- **friendly**

L'autorisation par défaut s'applique aux classes, interfaces, variables et méthodes.

Les éléments qui disposent de cette autorisation sont accessibles à toute les méthodes des classes du même package. Les classes dérivées ne peuvent donc y accéder que si elles sont explicitement déclarées dans le même package.

Rappelons que les classes n'appartenant pas explicitement à un package appartiennent automatiquement au package par défaut. Toute classe sans indication de package dispose donc de l'autorisation d'accès à toutes les classes se trouvant dans le même cas.

- **private**

L'autorisation *private* est la plus restrictive. Elle s'applique aux membres d'une classe (variables, méthodes, classes internes).

Les éléments déclarés *private* ne sont accessibles que depuis la classe qui les contient ; il n'est visible que dans la classe où il est défini.

Ce type d'autorisation est souvent employé pour les variables qui ne doivent être lues ou modifiées qu'à l'aide d'un *accesseur* ou d'un *modificateur*. Les accesseurs et les modificateurs, de leur côté, sont déclarés *public*, afin que tout le monde puisse utiliser la classe.

Tableau des modificateurs

Package 1	**Package2**
Class A	Class D extends A{}
	Class E {}

```
{int a ; public int b ;
protected int c ;
private int d ;
privateprotected int e ;
}
class B extends A
{}
class C{}
```

	Acces pour toutes les classes du package	Vu par tous	Pour les héritiers et les classes du même package	Uniquement dans la classe	Uniquement pour les héritiers
A partir de	**Friendly**	**public**	**protected**	**private**	**Private protected**
B extends A	**OUI**	**OUI**	**OUI**	**NON**	**OUI**
C	**OUI**	**OUI**	**OUI**	**NON**	**NON**
D extends A	**NON**	**OUI**	**OUI**	**NON**	**OUI**
E	**NON**	**OUI**	**NON**	**NON**	**NON**

Les classes abstraites, les interfaces, le polymorphisme

Le mot clé *abstract*

- **Méthodes et classes abstraites**

Une méthode déclarée *abstract* ne peut être exécutée. En fait, elle n'a pas d'existence réelle. Sa déclaration indique simplement que les classes dérivées doivent la redéfinir.

Les méthodes *abstract* présentent les particularités suivantes :

- Une classe qui contient une méthode *abstract* doit être déclarée *abstract*.
- Une classe *abstract* ne peut pas être instanciée.
- Une classe peut être déclarée *abstract*, même si elle ne comporte pas de méthodes *abstract.*
- Pour pouvoir être instanciée, une sous-classe d'une classe *abstract* doit redéfinir toute les méthodes *abstract* de la classe parente.
- Si une des méthodes n'est pas redéfinie de façon concrète, la sous-classe est elle-même *abstract* et doit être déclarée explicitement comme telle.
- Les méthodes *abstract* n'ont pas d'implémentation. Leur déclaration doit être suivie d'un point-virgule.

Ainsi dans l'exemple précédent la méthode *crie( )* de la classe *Animal* aurait pu être déclarée *abstract*, ce qui signifie que tout *Animal* doit être capable de crier, mais que le cri d'un animal est une notion abstraite. La méthode ainsi définie indique qu'une sous-classe devra définir la méthode de façon concrète.

```
abstract class Animal {

   // champs

   // méthodes

   abstract void crie() ;
}

class Chien extends Animal {

   // champs

   // méthodes

   void crie() {
         System.out.println("Ouah-Ouah !") ;
   }

}

class Chat extends Animal {

   // champs

   // méthodes

   void crie() {
         System.out.println("Miaou-Miaou !") ;
   }

}
```

De cette façon, il n'est plus possible de créer un animal en instanciant la classe *Animal*. En revanche, grâce à la déclaration de la méthode *abstract crie( )* dans la classe *Animal*, il est possible de faire crier un animal sans savoir s'il s'agit d'un chien ou d'un chat, en considérant les instances de Chien ou de Chat comme des instances de la classe parente.

```
Animal animal1 = new Chien() ;
Animal animal2 = new Chat() ;

animal0.crie() ;
animal1.crie() ;
```

Le premier animal crie "Ouah-Ouah !" ; le second, "Miaou-Miaou !".

Les interfaces

Une classe peut contenir des méthodes *abstract* et des méthodes non *abstract*. Cependant, il existe une catégorie particulière de classes qui ne contient que des méthodes *abstract*. Il s'agit des interfaces. Les interfaces sont toujours *abstract*, sans qu'il soit nécessaire de l'indiquer explicitement. De la même façon, il n'est pas nécessaire de déclarer leurs méthodes *abstract*.

Les interfaces obéissent par ailleurs à certaines règles supplémentaires.

- Elles ne peuvent contenir que des variables *static* et *final*.

- Elles peuvent être étendues comme les autres classes, avec une différence majeure : une interface peut dériver de plusieurs autres interfaces. En revanche, une classe ne peut pas dériver uniquement d'une ou de plusieurs interfaces. Une classe dérive toujours d'une autre classe, et peut dériver, en plus, d'une ou plusieurs interfaces.

Interface Point {…}

Class Point3D implements Point, implements …., implements… {…}

Casting

- **Sur-casting**

Un objet peut être considéré comme appartenant à sa classe ou à une classe parente selon le besoin, et cela de façon dynamique. Nous rappelons ici que toutes classes dérive de la classe *Object,* qui est un type commun à tous les objets. En d'autres termes, le lien entre une classe et une instance n'est pas unique et statique. Au contraire, il est établi de façon dynamique, au moment où l'objet est utilisé. C'est la première manifestation du polymorphisme !

Le sur-casting est effectué de façon automatique par Java lorsque cela est nécessaire. On dit qu'il est implicite. On peut l'expliciter pour plus de clarté, en utilisant l'opérateur de casting :

```
Chien chien = new Chien() ;
Animal animal = (Animal)chien ;
```

Après cette opération, ni le handle *chien,* ni l'objet correspondant ne sont modifiés. Les handles ne peuvent jamais être redéfinie dans le courant de leur existence. Seule la nature du lien qui lie l'objet aux handles change en fonction de la nature des handles.

Le sur-casting est un peu moins explicite, lorsqu'on affecte un objet à un handle de type différent. Par exemple :

```
Animal animal = new Chien() ;
```

Polymorphisme

- **Utilisation du sur-casting**

Considérons l'exemple suivant, qui reprend les définitions précédentes des classes *Animal, Chien,* et *Chat* et illustre une première sorte de polymorphisme avec sur-casting implicite sur la méthode *crie( )*.

```
public class Main {

   // méthodes

   public static void main(String[] argv) {
         Chien chien = new Chien() ;
         Chat chat = new Chat() ;
         crie(chien) ;
         crie(chat) ;

   void crie(Animal animal) {
         animal.crie() ;
   }
```

```
}
```

Le premier animal crie "Ouah-Ouah !" ; le second, "Miaou-Miaou !".

- ***Late-binding***

Il existe un moyen d'éviter le sous-casting explicite en Java, appelé *late-binding*. Cette technique fondamentale du polymorphisme permet de déterminer dynamiquement quelle méthode doit être appelée.

Dans la plupart des langages, lorsque le compilateur rencontre un appel de méthode, il doit être à même de savoir exactement de quelle méthode il s'agit. Le lien entre l'appel et la méthode est alors établi à la compilation. Cette technique est appelée *early binding* (liaison précoce). Java utilise cette technique pour les appels de méthodes déclarées *final*. Elle a l'avantage de permettre certaines optimisations.

En revanche, pour les méthodes qui ne sont pas *final*, Java utilise la technique du *late binding* (liaison tardive). Dans ce cas, le compilateur n'établit le lien entre l'appel et la méthode qu'au moment de l'exécution du programme. Ce lien est établi avec la version la plus spécifique de la méthode et doit être différencié du concept *abstract*. Considérons l'exemple suivant pour s'en convaincre.

```
class Animal {

   // méthodes

   void crie() {
   }
}

class Chien extends Animal {

   // méthodes

   void crie() {
         System.out.println("Ouah-Ouah !") ;
   }

}

class Chat extends Animal {

   // méthodes

   void crie() {
         System.out.println("Miaou-Miaou !") ;
   }
}

public class Main {

   // méthodes

   public static void main(String[] argv) {
         Chien chien = new Chien() ;
         Chat chat = new Chat() ;
         crie(chien) ;
         crie(chat) ;

   void crie(Animal animal) {
         animal.crie() ;
```

```
        }
}
```

Le premier animal crie "Ouah-Ouah !" ; le second, "Miaou-Miaou !". La méthode *crie* appelé dans la méthode *crie* de la classe *Main* est bien la plus spécifique, celle de *Chien* ou de *Chat* et non celle de *Animal* !

- ***Polymorphisme***

Le programme ci-dessous illustre le concept du polymorphisme. La classe *Animal* utilise la méthode abstraite *qui* pour définir la méthode *printQui* de manière plus ou moins abstraite. Cela entraîne une factorisation du code.

```
abstract class Animal {

   // méthodes

   abstract String qui() ;

   void printQui() {
         System.out.println("cet animal est un " + qui()) ;
   }
}

class Chien extends Animal {

   // méthodes

   String qui() {
         return "chien" ;
   }
}

class Chat extends Animal {

   // méthodes

   String qui() {
         return "chat" ;
   }
}
```

L'implémentation de la méthode *qui* est relié à l'appel, uniquement au moment de l'exécution, en fonction du type de l'objet appelant et non celui du handle !
C'est-à-dire,

```
Animal animal1 = new Chien() ;
Animal animal2 = new Chat() ;
animal1.printQui() ;
animal2.printQui() ;
```

donne comme résultat :

```
cet animal est un chien
cet animal est un chat
```

Spécificités du langage

Programme principal : la méthode *main*

(…)

Cette méthode doit impérativement être déclaré public. Ainsi la classe contenant la méthode *main( )*, le programme principal, doit être *public*. Rappelons ici qu'un fichier contenant un programme Java ne peut contenir qu'une seule définition de classe déclarée *public*. De plus le fichier doit porter le même nom que la classe, avec l'extension *.java*.

```
Class MonProg
{

Public static void main(String argv[])
{…}

}
```

Package

Les packages

- **Les packages accessibles par défaut**

La bibliothèque standard de Java est distribuée dans un certain nombre de packages, y compris java.lang, java.util, java.net, etc….Les packages standard de Java constituent des packages hiérarchiques. Tout comme les répertoires d'un disque dur, les packages peuvent être organisés suivant plusieurs niveaux d'imbrication. Tous les packages standard de Java se trouvent au sein des hiérarchies de packages java et javax.

- **L'instruction *package***

Une classe peut utiliser toutes les classes de son propre package et toutes les classes publiques des autres packages.
Vous pouvez accéder aux classes publiques d'un autre package de deux façons.

La première façon consiste simplement à ajouter le nom complet du package devant chaque nom de classes.

Exemple :
Java.util.Date today=new java.util.Date() ;

La seconde est d'utiliser l'instruction import

- **L'instruction *import***

Vous pouvez importer une classe spécifique ou l'ensemble d'un package. En plaçant l'instruction import en tête de vos fichiers source.
Exemple :

```
Import java.util.* ;

Date today=new Date() ;
```

Sans le préfixe du package, il est également possible d'importer une classe spécifique d'un package.

Import java.util.Date ;

Les tableaux

Les tableaux sont des objets, il faut utiliser l'opérateur NEW :

Exemple de déclaration et d'utilisation :

Int T[], int[]T ;
T=new int[10] ;

Ou int []T=new int[10] ;

Il existe une méthode length qui donne la longueur du tableau.

Int longueur=int.length nb T.length est interdit d'utilisation.

Les tableaux multidimensionnels.

Int M[][]
M=new int[10][20] ;

On peut initialiser un tableau lors de sa déclaration sans utiliser new

Exemple int[] T={3,4,2,1} ici new est inutile et implicite.

Les chaînes de caractères

Les chaînes de caractères sont des objets également.

Il existe deux classes :
String Chaîne constante (Non modifiable)
StringBuffer (Chaîne que l'on peut modifier)

Constructeur de String :

String ch1=new String(« Bonjour ») ;
String ch1= »Bonjour » ;

Quelques méthode sur la classe String (méthode statique)

Static String valueof(int i) ;
Renvoie la chaîne du type donnée.

Exemple : String valueof(12) => « 12 »

Boolean equals (String s) ;
Compare deux chaînes.

String concat(String s) ;
Concaténation de deux chaînes.

Int length() ;
Retourne la longueur de la chaîne de caractères.

Int indexof (int c) ;
Renvoie un entier correspondant à l'occurrence du caractère.

Char charAt(int i) ;
Renvoie le caractère de l'indice i.

Quelques méthode de la classe StringBuffer

Les constructeurs :

StringBuffer()
StringBuffer(int taille)
StringBuffer(String s)

StringBuffer append(String s) ; Ajouter une chaîne de caractères
StringBuffer append (char c) ; Ajouter un caractère

Int length() ; renvoie la longueur de la chaîne

String toString() ; Renvoie une chaîne constante de l'objet courant.

Les classes utilitaires

Vector :

C'est un tableau dynamique

Vector v=new Vector() ;

Il peut contenir un nombre quelconque d'objets, mais pas de type simple.

Quelques méthodes :

Void AddElement(object) ;

Void InsertElementAt(object,int) ;

Void SetElementAt(object,int) ;

Void RemoveElement(object) ;

Void RemoveElementAt(int) ;

Void RemoveAllElements() ;

Boolean Contains(object) ;

Object ElementAt(int);

Int IndexOf(object) ;

Int size() ;

Enumeration elements() ;

Exemple d'utilisation :

```
Vector T=new Vector() ;
For (Enumeration e=T.Elements() ;e.HasMoreElements() ;)
System.out.println(e.nextElement()) ;
```

Enumarations :

Représente une liste d'éléments, c'est une interface et non pas une classe, elle possède deux méthodes.

Enumeration E=new Enumeration() ;

Boolean HasMoreElements() ;

Object nextElement() ;

Hashtable :

Table de stockage permettant des recherches rapide, c'est un tableau à deux entrées.

Hashtable H=new Hashtable() ;

Object Put(clé,donnée) ;

Object Get(clé) ;

Object Remove(clé) ;

Boolean ContainsKey(clé) ;

EnumeratEnumeration

Enumeration Elements() ;

Les Applets

Une applet est un programme qui s'exécute sur une page Web.

Nb : Commentaire à ajouter dans le fichier source java pour éviter de faire une page HTML, ce qui permet d'utiliser l'utilitaire appletviewer.exe pour exécuter notre applet.

Exemple : appletviewer monapplet.java

Pour cela il faut introduire en commentaire la ligne suivante :

//<applet code= »MonApplet.class » height=400 width=400></applet>

Il n'y a pas de fonction main dans une applet.

Il y a par contre les méthodes suivantes à surcharger obligatoirement :

Void init() {…} Exécuté au début du programme.

Void start(){…} Elle s'exécute après init à chaque fois que l'appel à été stoppé et relancé.

Void stop(){…} S'éxécute quand l'applet est stoppée.

Void destroy(){…} S'éxécute à la destruction du programme.

Void paint(){…} Permet de dessiner sur la feuille de l'applet.

Dans l'exemple suivant on utilise l'objet (Graphics g).

Repaint() permet de réafficher l'écran.

```
Public class MonApplet extends Applet
{
Paint(Graphics g)
{
 g.drawString(« Bonjour »,10,5) ; //Ecrire un texte à l'écran
 g.drawline(x1,y1,x2,y2) ; //dessine une ligne
g.drawOvall(x,y,u,v) ; ou fillOvall // dessine un cercle  ou un ovale
g.drawRect(x1,y1,x2,y2) ; ou fillRect dessine un rectangle ou un carré
g.setColor(Color blue) ; affecte une couleur par défaut
g.setBackground(Color yellow) ; Affecte une couleur de fond
```

this.getSize() ; renvoie la dimension de l'objet courant

```
Dimension d=this.getsize() ;
```

On peut utiliser d.height ; et d.width par exemple…

Les threads

Les *threads* (en français processus indépendants) sont des mécanismes importants du langage Java. Ils permettent d'exécuter plusieurs programmes indépendants les uns des autres. Ceci permet une exécution parallèle de différentes tâches de façon autonome.

Un *thread* réagit aux différentes méthodes suivantes :

- *destroy( )* : arrêt brutal du *thread* ;
- *interrupt( )* permet d'interrompre les différentes méthodes d'attente en appelant une exception ;
- *sleep( )* met en veille de *thread* ;
- *stop( )* : arrêt non brutal du *thread* ;
- *suspend( )* : arrêt d'un *thread* en se gardant la possibilité de le redémarrer par la méthode *resume( ) ;*
- *wait( )* met le *thread* en attente ;
- *yield( )* donne le contrôle au schéduleur.

La méthode *sleep( )* est souvent employée dans les animations, elle permet de mettre des temporisations d'attente entre deux séquences d'image par exemple.

Les exceptions (*exception*) et les erreurs (*error*)

- **Deux types d'erreurs en Java**

En Java, on peut classer les erreurs en deux catégories :
- *les erreurs surveillées*,
- *les erreurs non surveillées*.

Java oblige le programmeur à traiter les erreurs surveillées. Les erreurs non surveillées sont celles qui sont trop graves pour que le traitement soit prévu à priori., comme par exemple la division par zéro.

```
Public class Erreur {
      Public static void main (String[] args) {
            int x = 10, y = 0, z = 0;
            z = x / y ;
      }
}
```

Dans ce cas, l'interpréteur effectue *un traitement exceptionnel*, il arrête le programme et affiche un message :

```
Exception in thread "main"
java.lang.ArithmeticException : / by zero
      at Erreur.main(Erreur.java:5)
```

- **Principe**

Lorsqu'une erreur de ce type est rencontrée, l'interpréteur crée immédiatement un objet, instance d'une classe particulière, elle-même sous-classe de la classe ***Exception***. Cet objet est créé normalement avec l'opérateur *new*. Puis l'interpréteur part à la recherche d'une portion de code capable de recevoir cet objet et d'effectuer le traitement approrié.

S'il s'agit d'une *erreur surveillée* par le compilateur, celui-ci a obligé le programmeur à fournir ce code. Dans le cas contraire, le traitement est fourni par l'interpréteur lui-même. Cette opération est appelée *lancement d'une exception* (***throw***). Pour trouver le code capable de traiter l'objet, l'interpréteur se base sur le type de l'objet, c'est-à-dire sur la classe dont il est une instance.

Pour reprendre l'exemple de la division par zéro, une instance de la classe `ArithmeticException` est lancée. Si le programme ne comporte aucun bloc de code capable de traiter cet objet, celui-ci est attrapé par l'interpréteur lui-même. Un message d'erreur est alors affichée `Exception in thread "main".`

- **Attraper les exceptions**

Nous avons vu que Java n'oblige pas le programmeur à attrapper tous les types d'exceptions. Seules celles correspondant à des *erreurs surveillées* doivent obligatoirement être attrapées. En fait, les exceptions qui ne peuvent pas être attrapées sont des instances de la classe ***RuntimeException*** ou une classe dérivée de celle-ci.

le multithreading

Définition

Un thread (appelé aussi processus léger ou activité) est une suite d'instructions à l'intérieur d'un process. Les programmes qui utilisent plusieurs threads sont dits multithreadés.

Syntaxe

Les `threads` peuvent être créés comme instance d'une classe dérivée de la classe `Thread`. Ils sont lancés par la méthode `start`(), qui demande à l'ordonanceur de `thread` de lancer la méthode `run`() du `thread`. Cette méthode `run`() doit être implantée dans le programme.

Premier exemple

```
class DeuxThreadAsynchrones {
public static void main(String args[ ]) {
new UneThread("la thread 1").start();
new UneThread("la seconde thread").start();
}
}
class UneThread extends Thread {
public UneThread(String str) {
super(str);
}
public void run() {
for (int i=0; i<10; i++) {
System.out.println(i+" "+getName());
try {sleep((int)(Math.random()*10));}
catch (InterruptedException e){}
}
System.out.println(getName()+" est finie");
}
}
```

une exécution

% java DeuxThreadAsynchrones

```
0 la thread 1
0 la seconde thread
1 la thread 1
2 la thread 1
1 la seconde thread
3 la thread 1
4 la thread 1
5 la thread 1
6 la thread 1
7 la thread 1
2 la seconde thread
3 la seconde thread
4 la seconde thread
```

8 la thread 1

5 la seconde thread

```
9 la thread 1
6 la seconde thread
la thread 1 est finie
7 la seconde thread
8 la seconde thread
```

```
9 la seconde thread
la seconde thread est finie
```

Une classe "threadée" C'est une classe qui implémente un `thread`. Syntaxiquement, on la construit :

- soit comme classe dérivée de la classe `Thread`.

Par exemple

```
class MaClasseThread extends Thread {
•••
}
```

- soit comme implémentation de l'interface `Runnable`.

```
class MaClasseThread implements Runnable {
•••
}
```

Cette dernière solution est la seule possible pour une `applet` "threadée" puisque Java ne supporte pas l'héritage multiple :

```
public class MonAppletThread extends Applet implements Runnable {
•••
}
```

constructeurs de `thread` Il y en a plusieurs. Quand on écrit

`t1 = new MaClasseThread();` le code lancé par `t1.start();` est le code `run`() de la classe threadée `MaClasseThread`.

Si on écrit :

```
t1 = new Thread(monRunnable);
```

le code lancé par `t1.start`(); est le code `run`() de la classe `monRunnable`. En général on a cette seconde syntaxe pour les applets qui implantent la méthode `run`() et la première syntaxe pour les applications Java.

Exemple :

```
class Appli extends Thread {
Thread t1;
public static void main(String args[]) {
t1 = new Appli();
t1.start();
}
public void run() { •••
// ce code est lancé
}
}
```

et

```
class MonApplet extends Applet implements Runnable {
public void init() {
Thread t1 = new Thread(this);
t1.start();
}
public void run() { •••
// ce code est lancé
}
}
```

le multithreading dans les applets

Il est bien de prévoir dans une `applet`, l'arrêt de traitement "désagréable" (animation ou musique intempestive) par clics souris. Il suffit d'écrire :

```
boolean threadsuspendue = false;
public boolean mouseDown(Event e, int x, int y) {
if (threadsuspendue) {
lathread.resume();
} else {
lathread.suspend();
}
threadsuspendue = !threadsuspendue;
return true;
}
```

Méthodes stop() et start()

Quand un browser est iconifié ou qu'il change de page, la méthode `stop`() de l'applet est lancée. Celle ci doit libérer les ressources, entre autre le processeur.
Si aucun `thread` n'a été implanté dans `l'applet`, le processeur est libéré.
Sinon il faut (sauf bonne raison) arrêter les `threads` en écrivant :

```
public void stop() {
if (lathread != null) {
lathread.stop();
lathread = null;
}
}
```

Pour relancer un thread arrété par `stop`(), il faut implanter la méthode `start`() par :

```
public void start() {
if (lathread == null) {
lathread = new Thread (•••);
lathread.start(); // ce N'est Pas recursif. OK ?
}
}
```

Passage de paramètres

```
class AutoThreadParam implements Runnable{
private Thread local;
private String param;
public AutoThreadParam(String param){
this.param = param;
local = new Thread( this);
local.start();
}
public void run(){
if( local == Thread.currentThread()){
while(true){
System.out.println("dans
AutoThreadParam.run"+ param);
}
}
}
}

public class Thread4 {
public static void main(String args[]) {
AutoThreadParam auto1 = new AutoThreadParam("auto 1");
AutoThreadParam auto2 = new AutoThreadParam ("auto 2");
```

```
while(true){
System.out.println("dans Thread4.main");
}
}
}
```

Les états d'un thread

À l'état "né", un `thread` n'est qu'un objet vide et aucune ressource système ne lui a été allouée. On passe de l'état "né" à l'état "`Runnable`" en lançant la méthode `start`(). Le système lui alloue des ressources, l'indique à l'ordonanceur qui lancera l'exécution de la méthode `run`(). A tout instant c'est un `thread` éligible de plus haute priorité qui est en cours d'exécution.

On entre dans l'état "`Not Runnable`" suivant 4 cas :

- quelqu'un a lancé la méthode `suspend`()
- quelqu'un a lancé la méthode `sleep`()
- le `thread` a lancé la méthode `wait`() en attente qu'une condition se réalise.
- le `thread` est en attente d'entrées/sorties

Pour chacune de ces conditions on revient dans l'état "`Runnable`" par une action spécifique :

- on revient de `sleep`() lorsque le temps d'attente est écoulé
- on revient de `suspend`() par `resume`()
- on revient de `wait`() par `notify`() ou `notifyAll`() exécutés par un autre `thread`..
- bloqué sur une E/S on revient a "`Runnable`" lorsque l'opération d'E/S est réalisé.

On arrive dans l'état "Mort" lorsque l'exécution de `run`() est terminée ou bien après un appel à `stop`().

méthodes publiques de "Thread"

```
final void suspend();
final void resume();
static native void yield();
final native boolean isAlive();
final void setName(String Nom);
final String getName();
public void run();
```

Concurrence

synchronisation en Java usage du mot-clé `synchronized`
Utilisation de `synchronized`

1er cas :

```
synchronized(obj){
// ici le code atomique sur l 'objet obj
}
```

2ieme cas
`synchronized void p(){ ……}`

qui signifie :

```
void p(){
synchronized (this){……}
}
```

Dans `java.lang.Object`

```
final void wait() throws InterruptedException
final native void wait(long timeout)
throws InterruptedException
…
```

Notifications

```
final native void notify()
final native void notifyAll()
wait(), notify() de la classe Object
```

remarque fondamentale
Ce sont des méthodes de la classe `Object` et non pas de la classe `Thread`, elles implantent des mécanismes de demande de verrou et d'avertissement pas `d'attente ( wait()` de `Object wait()` de `Thread)`. `notify()` (respectivement `notifyAll`()) avertit une des `threads` qui attend (respectivement toutes les `threads` qui attendent) sur un `wait`() sur l'objet sur lequel a été lancé `notify`() et débloque le `wait`() sur cet objet. Il faut donc avant de faire un `notify`() ou `notifyAll`() changer la condition de boucle qui mène au `wait`().

La méthode `wait`() relâche le moniteur de l'objet sur lequel elle a été lancée (sinon tout serait bloqué !!).
Ces deux méthodes doivent être lancées dans des méthodes `synchronized`.

Le mot clé réservé `synchronized` utilisé comme modifieur de méthode de classe ou d'instance (cf. ci dessus) :

```
public synchronized void methAtomique() { ••• }
```
ou
```
public static synchronized void methClasseAtomique() { ••• }
```

`synchronized` peut aussi être utilisé pour un bloc d'instructions :

```
synchronized (expression) { ••• }
```

où `expression` repère un objet.

synchronized : modifieur de méthode d'instance

Lorsqu'un `thread` veut lancer une méthode d'instance `synchronized`, le système Java pose un verrou sur l'instance. Par la suite, un autre `thread` invoquant une méthode `synchronized` sur cet objet sera bloqué jusqu'à ce que le verrou soit levé.
On implante ainsi l'exclusion mutuelle entre méthode `synchronized`. Attention les méthodes non `synchronized` ne sont pas en exclusion.

Synchronized pour un bloc

Dans `synchronized (expression) { ••• }`

où expression repère un objet, le système Java pose un verrou sur cet objet pendant l'exécution du bloc. Aucun autre bloc `synchronized` sur cet objet ne peut être exécuté et un tel bloc ne peut être lancé qu'après avoir obtenu ce verrou.

On définit ainsi une section critique

exemple :

```
public void trie(int[] tableau) {
synchronized (tableau) {
// votre algorithme de tri préféré
}
}
```

remarque
En fait

```
synchronized void meth() {•••}
```

signifie

```
void meth() {synchronized (this) {•••}}
```

Les entrées et sorties en Java

I/O Streams

Pour accéder à une information donnée, un programme ouvre un flux vers la source d'information (un fichier, mémoire etc.) et lie cette information de manière séquentielle. De la même manière que la lecture, pour écrire une information donnée, un programme ouvre un flux vers la sortie et là aussi, il écrit de manière séquentielle cette information sur le support d'écriture (fichier, mémoire etc.)

Les étapes de lecture/écriture sont identiques et se résument comme suit:

Lecture Écriture

```
Ouvre un flux en lecture
Lit tant qu'il y a quelque
chose à lire
Ferme le flux
Ouvre un flux en écriture
Écrit tant qu'il y a quelque
chose à écrire
Ferme le flux
```

Trois objets de flux sont crées automatiquement:

```
System.in (lecture)
System.out (écriture)
System.err (erreur standard, écriture)
```

Ces flux se trouvent dans le paquetage : `java.lang.System`

À ces flux là, vont s'ajouter une série de classes qui gèrent des flux de données. Ces classes se trouvent dans le paquetage `java.io.*` qui contient toutes les classes relatives à la gestion des flux entrée/sortie (autre que ceux de System). Pour pouvoir utiliser ces classes, il faudra les importer dans le fichier, comme suit:

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

Les flux de données sont divisés en 2 catégories, basées sur le type de données : binaire ou texte.

- **Flux text**e: sert à lire/écrire des informations textuelles codées sur 16 bits (donc `unicode` compris). Contient les superclasses: `Reader` et `Write`r.

- **Flux binair**e: sert à lire/écrire des informations codées en iso-latin-1 sur 8 bits. Il Contient les superclasses: `InputStream` et `OutputStrea`m.
Ces deux flux contiennent grosso modo les mêmes méthodes, sauf qu'elles sont spécifiques à chacun des flux (texte ou binaire):

`Reader:`
```
int read();
int read(char cbuf[]);
int read(char cbuf[], int offset, int length);
```

`Writer:`
```
int write(int c);
int write(char cbuf[]);
int wrtie(char cbuf[], int offset, int length);
```

`InputStream:`
int read();
int read(byte cbuf[]);
int read(byte cbuf[], int offset, int length);

`OutputStream:`
int write(int c); écrit l'octet de poids faible de c
int write(byte cbuf[]);
int write(byte cbuf[], int offset, int length);

Quelques types flux I/O

`Buffering(BufferdReader/BufferdWriter ; BufferdInputStream/BufferdOutputStream)`:
une zone tampon, pour réduire les accès en lecture/écriture

`DataConversion(DataInputStream/DataOutputStream)`: lecture/écriture dans un format indépendant de la machine.

`File(FileReader/FileWriter ; FileInputStream/FileOutputStream)`: lire et écrire à partir de fichiers

`Printing(PrintWriter ; PrintOutputStream)`: contient des méthodes données pour imprimer suivant un format donné.

Remarque :

Faire attention lors de la lecture/écriture textuelle. L'écriture suivante est sans intérêt mais permise car PrinterWriter a déjà un buffer:

```
PrintWriter sortie = new PrintWriter
(new BufferedWriter
(new FileWriter("ficsortie.txt")));
```

Mais vu qu'il n'y a pas d'équivalent à PrintWriter pour PrintReader et vu que Print est associé à l'écriture et non pas à la lecture, il faut donc bufférisér, car FileReader se contente de lire uniquement les caractères, et si on veut aussi les retour de lignes, il faut utiliser BufferWriter, ainsi:

```
BufferedReader = new BufferReader (new FileReader("entree.txt")));
```

Utilisation d'objet de type File

Cette classe offre la possibilité de gestion de fichiers (créer, renommer, détruire etc.) et aussi de répertoires (créer, renommer, détruire etc.).

Création d'un objet de type File

```
File monfic = new File ("unfic.txt");
File monfic = new File ("c:\\test\\monrep");
```
Pour ouvrir un flux en entrée vers le fichier unfic.txt
```
FileReader infic = new FileReader(monfic);
```

ou bien en une seule opération:

```
FileReader infic = new FileReader(new File ("unfic.txt"));
```

L'intérêt de faire cela est de profiter des méthodes qui se trouvent dans la classe File.
Sinon pas la peine de le faire.

Si on tient compte du Buffer, cela devient:

```
BufferedReader = new BufferReader (new FileReader(new File("unfic.txt"))));
```

La sérialisation dans Java

L'un des fondements de la programmation Orientée Objet réside dans l'idée de réutilisation. La plate forme Java collecte pour sa part les objets en mémoire afin qu'ils puissent être accessibles à partir de n'importe quelle application Java. Cependant, cette réutilisabilité n'est valable que tant que la *Java Virtual Machine* (JVM) est lancée : si celle-ci est arrêtée, le contenu de la mémoire disparaît.

C'est ici que la sérialisation intervient : elle permet de stocker l'état des objets, ainsi que la manière de recréer ces objets pour plus tard. Un objet peut ainsi exister entre deux exécutions d'un programme, ou entre deux programmes : c'est la persistance objet. Ces objets stockés sont appelés des données série, c'est à dire envoyée une par une (à la manière du port série d'un ordinateur) et non simultanément (comme le port parallèle).

Le objets peuvent être sérialisés sur le disque dur interne, ou sur le réseau disponible (Internet compris). Pour rendre un objet persistant, il faut le sérialiser en implémentant l'interface `java.io.Serializable` au sein de la classe : Java saura alors que faire de l'objet.

```
import java.io.*;
import java.util.Date;
import java.util.Calendar;

public class Serialisation
  {
  public static void main(String [] args) throws IOException
    {
    PersistentTime time = new PersistentTime();
    FileOutputStream f = FileOutputStream("time");
    ObjectOutputStream o = ObjectOutputStream(f);
    o.writeObject(time);
    o.close();
    }
  }

public class PersistentTime implements Serializable
  {
  private Date time;
  public PersistentTime()
    {
    time = Calendar.getInstance().getTime();
    }
  public Date getTime()
    {
    return time;
    }
  }
```

Le programme crée alors une instance de `PersistentTime`, modifie la valeur time puis sérialise l'objet à l'aide d'un flux `ObjectOutputStream` enveloppé dans un `FileOutputStream`. L'objet sérialisé est alors enregistré sur le disque dans un fichier nommé time. C'est l'appel à la méthode `o.writeObject()` qui lance la sérialisation. On obtient un fichier nommé "`time`".

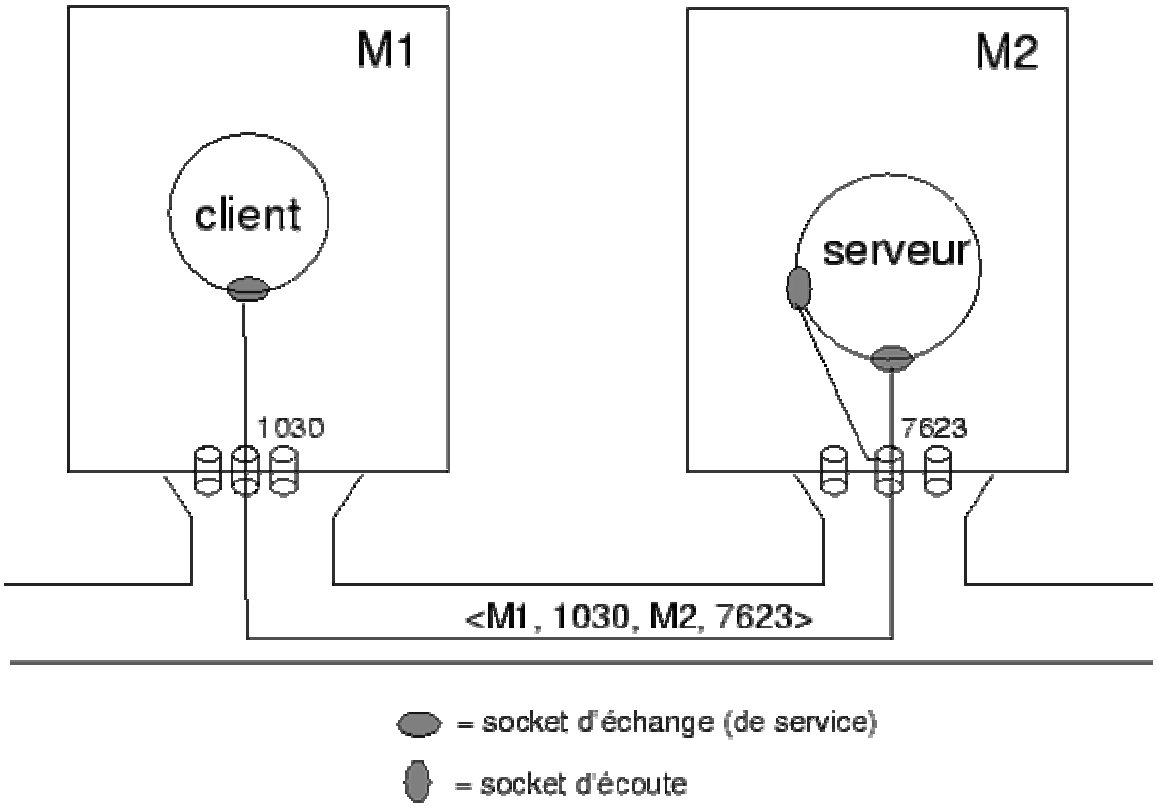
Pour récupérer l'objet, il faut alors passer par le code suivant :

```
import java.io.*;
import java.util.Date;
import java.util.Calendar;

public class Deserialisation
  {
  public static void main(String [] args) throws IOException
    {
    FileInputStream f = new FileInputStream("time");
    ObjectIntputStream o = ObjectInputStream(f);
    time = (PersistantTime)o.readObject();
    o.close();

    System.out.println("Donnée sauvée: " + time.getTime());
    System.out.println("Donnée            actuelle:            "            +
Calendar.getInstance().getTime());
    }
  }
```

Inversement, donc, c'est `o.readObject()` qui permet de lancer la désérialisation. On affiche ensuite la valeur stockée et la valeur en cours pour comparaison.

La communication par socket en Java

L'interface socket BSD est l'interface de programmation réseau la plus courante. Cette interface permet de façon classique d'utiliser une communication par socket selon le schéma habituel client-serveur. L'interface Java des sockets (package `java.net`) offre un accès simple aux sockets sur IP.



Les classes

Plusieurs classes interviennent lors de la réalisation d'une communication par sockets. La classe `java.net.InetAddress` permet de manipuler des adresses IP. La classe `java.net.SocketServer` permet de programmer l'interface côté serveur en mode connecté. La classe `java.net.Socket` permet de programmer l'interface côté client et la communication effective par flot via les sockets. Les classes `java.net.DatagramSocket` et `java.net.DatagramPacket` permettent de programmer la communication en mode datagramme.

La classe `java.net.InetAddress`

Cette classe représente les adresses IP et un ensemble de méthodes pour les manipuler. Elle encapsule aussi l'accès au serveur de noms DNS.

```
public class InetAddress implements Serializable
```

- **Les opérations de la classe `InetAddress`**

Conversion de nom vers adresse IP :
Un premier ensemble de méthodes permet de créer des objets adresses IP.

```
public static InetAddress getLocalHost() throws UnknownHostException
```

Cette méthode renvoie l'adresse IP du site local d'appel.

```
public static InetAddress getByName(String host) throws
UnknownHostException
```

Cette méthode construit un nouvel objet `InetAddress` à partir d'un nom textuel de site. Le nom du site est donné sous forme symbolique (`crabe.univ-valenciennes.fr`) ou sous forme numérique (***.***.**.**).

Enfin,

```
public static InetAddress[] getAllByName(String host) throws
UnknownHostException
```

permet d'obtenir les différentes adresses IP d'un site.

Conversions inverses
Des méthodes applicables à un objet de la classe `InetAddress` permettent d'obtenir dans divers formats des adresses IP ou des noms de site. Les principales sont :

`public String getHostName()`	obtient le nom complet correspondant à l'adresse IP
`public String getHostAddress()`	obtient l'adresse IP sous forme %d.%d.%d.%d
`public byte[] getAddress()`	obtient l'adresse IP sous forme d'un tableau d'octets.

La classe `ServerSocket`

Cette classe implante un objet ayant un comportement de serveur via une interface par socket.

```
public class java.net.ServerSocket
```

Une implantation standard du service existe mais peut être redéfinie en donnant une implantation explicite sous la forme d'un objet de la classe `java.net.SocketImpl`. Nous nous contenterons d'utiliser la version standard.

- **Le constructeur `ServerSocket`**

```
public ServerSocket(int port) throws IOException
public ServerSocket(int port, int backlog) throws IOException
public ServerSocket(int port, int backlog, InetAddress bindAddr) throws
IOException
```

Ces constructeurs créent un objet serveur à l'écoute du port spécifié. La taille de la file d'attente des demandes de connexion peut être explicitement spécifiée via le paramètre `backlog`. Si la machine possède plusieurs adresses, on peut aussi restreindre l'adresse sur laquelle on accepte les connexions.

Appels système :
ce constructeur correspond à l'utilisation des primitives `socket`, `bind` et `listen`.

- **Les opérations de la classe `ServerSocket`**

Nous ne retiendrons que les méthodes de base. La méthode essentielle est l'acceptation d'une connexion d'un client :

```
public Socket accept() throws IOException
```

Cette méthode est bloquante, mais l'attente peut être limitée dans le temps par l'appel préalable de la méthode `setSoTimeout`. Cette méthode prend en paramètre le délai de garde exprimé en millisecondes. La valeur par défaut 0 équivaut à l'infini. À l'expiration du délai de garde, l'exception `java.io.InterruptedIOException` est levée.

```
public void setSoTimeout(int timeout) throws SocketException
```

La fermeture du socket d'écoute s'exécute par l'appel de la méthode `close`.

Enfin, les méthodes suivantes retrouvent l'adresse IP ou le port d'un socket d'écoute :

```
public InetAddress getInetAddress()
public int getLocalPort()
```

La classe `java.net.Socket`

La classe `java.net.Socket` est utilisée pour la programmation des sockets connectés, côté client et côté serveur.

```
public class java.net.Socket
```

Comme pour le serveur, nous utiliserons l'implantation standard bien qu'elle soit redéfinissable par le développement d'une nouvelle implantation de la classe `java.net.SocketImpl`.

- **Constructeurs**

Côté serveur, la méthode `accept` de la classe `java.net.ServerSocket` renvoie un socket de service connecté au client. Côté client, on utilise :

```
public Socket(String host, int port) throws UnknownHostException,
IOException
public Socket(InetAddress address, int port) throws IOException
public Socket(String host, int port, InetAddress localAddr, int
localPort)
              throws UnknownHostException, IOException
public Socket(InetAddress addr, int port, InetAddress localAddr, int
localPort)
              throws IOException
```

Les deux premiers constructeurs construisent un socket connecté à la machine et au port spécifiés. Par défaut, la connexion est de type TCP fiable. Les deux autres interfaces permettent en outre de fixer l'adresse IP et le numéro de port utilisés côté client (plutôt que d'utiliser un port disponible quelconque).

Appels système :
ces constructeurs correspondent à l'utilisation des primitives `socket`, `bind` (éventuellement) et `connect`.

- **Opérations de la classe `Socket`**

La communication effective sur une connexion par socket utilise la notion de flots de données (`java.io.OutputStream` et `java.io.InputStream`). Les deux méthodes suivantes sont utilisés pour obtenir les flots en entrée et en sortie.

```
public InputStream  getInputStream()  throws IOException
public OutputStream getOutputStream() throws IOException
```

Les flots obtenus servent de base à la construction d'objets de classes plus abstraites telles que `java.io.DataOutputStream` et `java.io.DataInputStream` (pour le JDK1), ou `java.io.PrintWriter` et `java.io.BufferedReader` (JDK2) (cf exemple 2).

Une opération de lecture sur ces flots est bloquante tant que des données ne sont pas disponibles. Cependant, il

est possible de fixer une délai de garde à l'attente de données (similaire au délai de garde du socket d'écoute : levée de l'exception
`java.io.InterruptedIOException`):

```
public void setSoTimeout(int timeout) throws SocketException
```

Un ensemble de méthodes permet d'obtenir les éléments constitutifs de la liaison établie :

`public InetAddress getInetAddress()`	fournit l'adresse IP distante
`public InetAddress getLocalAddress()`	fournit l'adresse IP locale
`public int getPort()`	fournit le port distant
`public int getLocalPort()`	fournit le port local

L'opération `close` ferme la connexion et libère les ressources du système associées au socket.

Socket en mode datagramme `DatagramSocket`

La classe `java.net.DatagramSocket` permet d'envoyer et de recevoir des paquets (datagrammes UDP). Il s'agit donc de messages non fiables (possibilités de pertes et de duplication), non ordonnés (les messages peuvent être reçus dans un ordre différent de celui d'émission) et dont la taille (assez faible -- souvent 4Ko) dépend du réseau sous-jacent.

```
public class java.net.DatagramSocket
```

- **Constructeurs**

```
public DatagramSocket() throws SocketException
public DatagramSocket(int port) throws SocketException
```

Construit un socket datagramme en spécifiant éventuellement un port sur la machine locale (par défaut, un port disponible quelconque est choisi).

- **Émission/réception**

```
public void send(DatagramPacket p) throws IOException
public void receive(DatagramPacket p) throws IOException
```

Ces opérations permettent d'envoyer et de recevoir un paquet. Un paquet est un objet de la classe `java.net.DatagramPacket` qui possède une zone de données et (éventuellement) une adresse IP et un numéro de port (destinataire dans le cas `send`, émetteur dans le cas `receive`). Les principales méthodes sont :

```
public DatagramPacket(byte[] buf, int length)
public DatagramPacket(byte[] buf, int length, InetAddress address, int
port)

public InetAddress getAddress()
public int getPort()
public byte[] getData()
public int getLength()

public void setAddress(InetAddress iaddr)
public void setPort(int iport)
public void setData(byte[] buf)
public void setLength(int length)
```

Les constructeurs renvoient un objet pour recevoir ou émettre des paquets. Les accesseurs `get*` permettent, dans le cas d'un `receive`, d'obtenir l'émetteur et le contenu du message. Les méthodes de modification `set*` permettent de changer les paramètres ou le contenu d'un message pour l'émission.

Connexion
Il est possible de « connecter » un socket datagramme à un destinataire. Dans ce cas, les paquets émis sur le socket seront toujours pour l'adresse spécifiée. La connexion simplifie l'envoi d'une série de paquets (il n'est plus

nécessaire de spécifier l'adresse de destination pour chacun d'entre eux) et accélère les contrôles de sécurité (ils ont lieu une fois pour toute à la connexion). La « déconnexion » enlève l'association (le socket redevient disponible comme dans l'état initial).

```
public void connect(InetAddress address, int port)
public void disconnect()
```

- **Divers**

Diverses méthodes renvoient le numéro de port local et l'adresse de la machine locale (`getLocalPort` et `getLocalAddress`), et dans le cas d'un socket connecté, le numéro de port distant et l'adresse distante (`getPort` et `getInetAddress`). Comme précédemment, on peut spécifier un délai de garde pour l'opération `receive` avec `setSoTimeout`. On peut aussi obtenir ou réduire la taille maximale d'un paquet avec `getSendBufferSize`, `getReceiveBufferSize`, `setSendBufferSize` et `setReceiveBufferSize`.

Enfin, n'oublions pas la méthode `close` qui libère les ressources du système associées au socket.

Java RMI

Introduction

Nous avons expliqué le principe de la programmation réseau en Java. Programmer directement au niveau sockets TCP et UDP, peut s'avérer utile dans de nombreuses situations, surtout quand le programmeur souhaite implémenter son propre protocole applicatif.
Cependant dans de nombreux autres cas, et cela est de plus en plus nombreux avec l'émergence de l'informatique répartie dans les entreprises, il s'avère nécessaire de pouvoir accéder à des objets Java à distance.
Pour faciliter la tache du programmeur, les concepteurs de Java ont crée plusieurs classes permettant l'invocation de méthodes à distance : c'est le *Remote Method Invocation (RMI)*.
Avec RMI, toute la programmation au niveau socket est cachée, et le programmeur n'a à se soucier que de l'implémentation des objets distants de son programme.
Dans ce chapitre nous aborderons les principes de base de RMI en les illustrant par un exemple.

Vue d'ensemble

Une application RMI est composée d'une partie client et d'une partie serveur. Le rôle d'un serveur est de créer des objets qu'on qualifie de distants, de les rendre accessibles à distance et enfin d'accepter des connexions de clients vers ces objets. Le rôle d'un client est donc d'accéder aux méthodes des ces objets, tout en considérant ces objets comme des objets locaux.
RMI est le mécanisme qui permet d'assurer la communication entre le serveur et le client et la transparence des accès.

Le principe de fonctionnement d'une application RMI est le suivant :

- Le serveur crée les objets et les enregistre auprès d'un service appelé *rmiregistry*. Cela permettra aux clients de localiser les objets et d'obtenir des références vers ces objets.
- Lorsqu'un client désire invoquer un objet a distance, il consulte la *rmiregistry* pour localiser l'objet : il fournit le nom de l'objet et reçoit en retour une référence vers cet objet.
- Avec la référence nouvellement obtenue, le client pourra invoquer les méthodes de cet objet.

Commençons à entrer de plus en plus dans les détails de fonctionnement de RMI.
Un objet désirant être invoqué à distance (devenir un *remote object* en anglais), devra implémenter une interface qui étend l'interface java.rmi.Remote. Lorsqu'un client obtient une référence vers cet objet, il obtient en fait une référence vers un *stub*. Un *stub* est un mandataire de l'objet qui est chargé dans le client au moment de l'obtention de la référence. Ainsi la référence que le client utilise pointe vers ce mandataire qui d'ailleurs implémente les mêmes interfaces que l'objet distant. Le client invoque ainsi par le biais de sa référence les

méthodes qui résident dans le *stub* qui se chargent alors elles-mêmes d'invoquer les vraies méthodes sur l'objet distant en passant à travers le réseau. C'est ce mécanisme qui assure la transparence des appels.

Donc la création d'une application repartie en RMI consiste en les étapes suivantes :

- Concevoir et implémenter les composants de l'application sous forme d'objets locaux ou distants.
- Compiler les sources et générer les *stubs*.
- Rendre les objets accessibles à travers le réseau.
- Démarrer l'application.

Le reste de ce chapitre se concentre sur la présentation d'un exemple de base de programmation en RMI.
Notre programme fonctionne de la manière suivante : Un serveur représente une boutique en ligne. Dans cette boutique nous avons plusieurs produits. Les clients seront en mesure d'interroger le serveur pour connaître les produits disponibles et leurs prix.
Les paragraphes suivants correspondent aux phases suivantes :

- Création du serveur.
- Création du client.
- Compilation et exécution de l'application.

Le Serveur

Le serveur consiste en une interface qui définit les fonctionnalités du serveur que les clients pourront invoquer, et en une classe qui implémente cette interface.

L'interface du serveur

Dans l'interface `Boutique_Interface` nous définissons la méthode que le client pourra invoquer pour obtenir la liste des produits et leurs prix.

L'interface du serveur : Boutique_Interface

```
import java.rmi.Remote ;
import java.rmi.RemoteException ;

public interface Boutique_Interface extends Remote {
      public String produitSuivant() throws RemoteException ;
}
```

Ce qui est important de mémoriser c'est que notre interface devra implémenter l'interface `Remote` de RMI. En plus, chaque méthode de notre interface devra inclure dans sa déclaration le `throws RemoteException`.
En étendant l'interface `Remote`, notre interface indique que tout objet qui l'implémente devient un objet distant dont les méthodes peuvent être invoquées à distance.

La classe du serveur

En général, une classe qui implémente un objet distant devra effectuer les étapes suivantes :

- Indiquer l'interface qu'elle implémente.
- Implémenter les méthodes définies dans l'interface.

La classe du serveur : Boutique

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

public class Boutique extends UnicastRemoteObject implements
Boutique_Interface {

      public Boutique() throws RemoteException {
            super() ;
      }

      public String produitSuivant() {
      return("Bananes -- quantite : 2 -- prix unite : 0.30euros") ;
      }

      public static void main(String[] args) {
      if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
      String nom = "//localhost/Boutique";
        try {
            Boutique_Interface boutique = new Boutique();
            Naming.rebind(nom, boutique);

            System.out.println("association nom <--> objet : ok");
        } catch (Exception e) {
```

```
            System.err.println("Boutique exception: " +
                               e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Comme nous le constatons, notre classe implémente l'interface `Boutique_Interface` et étend la classe `UnicastRemoteObject`. Cette classe définit les méthodes de la classe `Object` afin de les rendre appropriées à l'invocation à distance. Cette méthode permet aussi au *`runtime RMI`* de prendre connaissance de notre nouvelle classe.
Dans le constructeur nous invoquons celui de la classe mère (`UnicastRemoteObject`) ce qui rend *`exportable`* notre objet : il pourra désormais recevoir des invocations de méthodes à distance en écoutant sur un certain port.
Ensuite nous implémentons la méthode qui permet de retourner le prix des produits de notre boutique, comme si nous implémentions une méthode ordinaire.
Comme dans tout programme Java, nous devons implémenter la méthode main. Dans cette méthode, la première chose que nous devons faire est de créer et d'installer un gestionnaire de sécurité. Ce gestionnaire permet de protéger notre machine virtuelle de tout utilisateur malintentionné qui essaye de faire tourner du code malicieux sur notre machine. Dans notre cas nous utilisons un gestionnaire de sécurité fournit avec les classes RMI : `RMISecurityManager`.
La dernière étape est de rendre notre classe accessible aux clients. Nous créons alors un objet de type Boutique par :
`Boutique_Interface boutique = new Boutique();`
Il est important de noter que notre nouvel objet est de type `Boutique_Interface` et non Boutique car ce qui doit être visible aux clients sont des objets de type `Boutique_Interface` et non du type Boutique. Ainsi seules les méthodes déclarées dans `Boutique_Interface` sont accessibles par les clients.
Java RMI fournit une classe `Naming` permettant d'enregistrer des objets distants et de faire une résolution de référence vers ces objets. Pour enregistrer notre objet auprès de la *`registry`* nous commençons par former un nom permettant d'identifier notre objet. Dans le nom figure le nom du serveur sur lequel se trouve l'objet (et la *registry*) et le nom par lequel on aimerai identifier notre objet au sein de cette *registry*. Après avoir formé le nom, nous invoquons la méthode rebind de la classe Naming, cela permet d'associer le nom de notre objet avec le nom choisi.

Le Client

La classe du client : Client

```
import java.rmi.*;

public class Client {
      public static void main(String[] args) {
            if (System.getSecurityManager() == null) {
                        System.setSecurityManager(new
RMISecurityManager());
            }
            try {
                        String nom = "//" + args[0] + "/Boutique";
                  Boutique_Interface    boutique    =    (Boutique_Interface)
Naming.lookup(nom);
                  System.out.println(boutique.produitSuivant());
            }
            catch (Exception e) {
                        System.err.println("Client     exception:     "     +
e.getMessage());
                        e.printStackTrace();
            }
      }
}
```

Comme dans le cas du serveur, le client commence par installer un gestionnaire de sécurité. Cela est nécessaire car RMI devra télécharger du code dans la machine virtuelle du client en l'occurrence le *stub*.
Ensuite le client construit un nom permettant de faire une résolution de référence vers l'objet distant. Le nom de la machine qui contient l'objet distant est passé en argument. Le client utilise la méthode `lookup` de la classe `Naming` pour réussir cette résolution de référence.
Notons que l'objet est de type `Boutique_Interface`.

Lancer le programme

Maintenant que nous avons vu le code du client et du serveur, procédons à faire tourner notre petit programme.
Côté serveur, il faudra compiler les fichiers .java : pour cela rien de plus simple qu'un `javac *.java`.
Ensuite nous devons procéder à la génération du *stub* correspondant à la classe Boutique. Ce *stub* sera transféré aux clients au moment de la résolution de la référence. Pour cela nous devrons utiliser la commande *rmic* (pour RMI compiler) : `rmic Boutique`.
Côté client avant de compiler la classe client, il faut s'assurer qu'on a la classe `Boutique_Interface.class` car notre client s'en sert de cette classe.
Après avoir tout compilé, nous procédons au lancement de notre programme.
Dans notre cas, nous allons lancer un serveur web qui nous permettra de télécharger les *stubs* du serveur vers le client à la demande.

Avant de lancer le serveur web il est important de noter que dans Java 2 (jdk1.2 et versions ultérieures) les politiques de sécurité appliquées par la JVM ne permettent pas du téléchargement du code sans permission explicite. Pour cela nous avons besoin, du côté client comme du côté serveur d'un fichier de politique de sécurité qu'on nommera `java.policy` et contenant les lignes suivantes :

```
grant {
    permission java.net.SocketPermission "*:1024-65535",
        "connect,accept";
    permission java.net.SocketPermission "*:80", "connect";
};
```

Pour le serveur web, un simple petit serveur suffit (un simple serveur existe sur ftp://ftp.javasoft.com/pub/jdk1.1/rmi/class-server.zip)

Le lancement se fait par :

`start java ClassServer 2001 /chemin_du_serveur` où `chemin_du_serveur` est le chemin du répertoire où les classes existent.

Ensuite, sur la machine serveur, il faut lancer la rmiregistry.

```
start rmiregistry
```

Attention: le lancement du `rmiregistry` devra se faire d'un endroit où les classes du serveur ne sont pas vues (c-à-d hors le classpath), sinon la `rmiregistry` ne passera pas par le serveur web pour envoyer les `stubs` vers le client.

Le lancement du serveur se fait par :

```
java -Djava.rmi.server.codebase=http://localhost:2001/
-Djava.rmi.server.hostname=localhost
-Djava.security.policy=java.policy Boutique
```

Le lancement du client se fait par :

```
java -Djava.security.policy=java.policy Client localhost
```

Si tout ce passe bien, nous aurons une réponse du serveur comme quoi l'association à la `registry` s'est bien passée, et du côté client une interrogation réussie du serveur.