Développer des applications avec le langage Java

15 juin 1999

Table des matières

1 Présentation du langage

1.1 Introduction

Le langage `java` a fait une entrée spectaculaire dans le monde du développement logiciel en 1995. Il combinait des options déjà popularisées par certains de ces concurrents (l'orientation objet) , et des caractéristiques particulièrement adaptées aux autres tendances dominantes du moment (Internet).

1.2 Philosophie

1.2.1 Un langage orienté objet

L'orientation objet s'est progressivement imposée comme technologie de développement, laissant espérer dans la conception des applications:

- plus de réutilisation (du code, mais aussi de l'architecture)
- une plus grande réactivité au changement
- un moindre coût de maintenance

Les langages orientés objets les mieux connus étaient `C++` et `SmallTalk`. `java` occupe une position intermédiaire, plus proche de `C++` en ce qui concerne le style de programmation. Nous verrons les principes de la programmation objet un peu plus loin, mais la notion essentielle est celle de "classe", qui définit le comportement des variables du langage, appelés objets, et un programme `java` est un ensemble de définition de classes.

1.2.2 Un langage interprété

`java` est un langage compilé, la compilation produisant un pseudocode destiné à être interprété par une machine virtuelle, habituellement désignée sous le nom de JVM. Les spécifications précises tant du peudocode que du modèle d'exécution de la JVM font partie intégrante des spécifications du langage.

1.2.3 Un langage multi-plateformes

Les spécifications ci-dessus étant indépendantes de tout environnement physique, le premier bénéfice est la portabilité: en principe, une application écrite *et* compilée dans un environnement peut être exécutée dans n'importe quel autre. Bien sûr quelques précautions sont à prendre:

- Un constructeur peut "respecter" les spécifications de la machine virtuelle tout en introduisant des extensions. Si une application tire partie de ces extensions, elle ne pourra fonctionner que dans les environnement les implémentant.
- Un programme "réaliste" doit procéder avec le monde extérieur (le système fichier, l'interface utilisateur, les bases de données). Les concepteurs du langage ont fait de leur mieux pour rendre portable cet interfaçage, mais il n'est pas toujours possible de "cacher" le problème au développeur qui a besoin des fonctionnalités spécifiques d'un environnement.
- Un seul langage n'étant pas toujours suffisant pour toutes les applications, certains programmes peuvent devoir utiliser l"'ouverture" de `java` pour appeler des fonctions extérieures (`native`). Ceci bien sûr rend le développeur responsable de la portabilité correspondante.

Malgré ces réserves, la portabilité est nettement supérieure à celle du `C`, or celle-ci, bien que plus imparfaite, était déjà assez bonne pour être un des facteurs de son succès. La portabilité de `java` est donc *très* satisfaisante.

1.2.4 Un environnement d'exécution complet

La machine virtuelle est prolongée par une bibliothèque de classes très complète qui non seulement accélère le développement des applications mais joue un rôle important dans leur portabilité. Ces classes sont regroupées en "packages".

1.3 `java` et Internet

En plus des bénéfices décrits dans les paragraphes précédents, la structure du pseudocode est telle qu'il est possible pour la machine virtuelle de requérir des éléments compilés (les `.class`), en fonction des besoins, auprès de n'importe quelle source. Ceci est évidemment très adapté au Web, et a donné naissance à la notion d'`applet`, mini-applications pouvant fonctionner dans un browser sans que l'utilisateur ait eu à se préocupper de leur installation, et qui ont été l'un des principaux facteurs de l'explosion médiatique de `java`, même si l'intérêt du langage est très loin de s'y limiter.

1.4 Le marché

La popularité de `java` a atteint des sommets littéralement inouïs pour un langage de programmation: le grand public peut voir son nom dans tous les

media! De façon plus concrète pour son succès industriel, beaucoup d'entreprises s'y sont intéressées:

- Du point de vue du style de développement, il remplit un créneau non encore occupé dans la gamme des langages orientés objet (il reste proche de `C++`, dont le succés, mesuré en nombre de developpeurs, est supérieur à celui de concurrents comme `SmallTalk`), et nous verrons qu'il protége le développeur des difficultés de plus bas niveau rencontrées dans `C++`.
- Ses possibilités d'adaptation à des environnements répartis, basés sur des composants, sont immédiates.
- La notion de langage interprété suscite toujours des inquiétudes en ce qui concerne les performances. S'il est vrai qu'une exécution interprétée ne peut pas rivaliser avec l'exécution de code machine, les machines virtuelles sont libres, une fois le pseudocode chargé et vérifié, de le traduire en language machine pour l'exécution ultérieure. Ce mécanisme, connu sous le nom de "compilation Just In Time" a beaucoup fait pour dédramatiser le problème.

2 L'orientation objet

2.1 Les principes généraux

Les 4 principaux mots magiques de la programmation objet sont: l'encapsulation l'abstraction, l'héritage, et le polymorphisme.

L'encapsulation: au lieu de s'intéresser principalement aux procédures, le développement objet cherche à regrouper un ensemble de données et de fonctions qui peuvent leur être appliquées. L'ensemble de ces données (attributs) et de ces fonctions (méthodes) constitue un objet, et, si le langage est typé, est décrit dans un type qu'on appelle une classe.

La combinaison de l'encapsulation et de l'abstraction conduit à ne présenter aux utilisateurs d'une telle classe que les méthodes d'intérêt public, les protégeant des détails de la réalisation, et de leur évolution.

L'héritage désigne la possibilité de construire de nouvelles classes en s'appuyant sur des classes existantes: la nouvelle classe (dérivée), hérite des attributs et des méthodes de l'ancienne (base).

Le polymorphisme désigne la possibilité d'écrire une seule fois du code qui pourra s'appliquer à des objets différents, en exécutant des méthodes différentes, mais avec la même signification. Il existe plusieurs formes de polymorphisme, certaines sans rapport avec l'orientation objet, mais la plus importante est réalisée à travers l'héritage, en autorisant la classe dérivée à *redéfinir* certaines méthodes de la classe de base. La différence entre l'héritage seul et l'héritage combiné avec le polymorphisme est à peu près la même qu'entre une mobylette et une moto de course, ne pas oublier qu'une moto de course peut être très dangereuse si elle est mal utilisée! D'une part l'héritage peut être utilisé "à tort", quand par exemple un objet devrait simplement posséder une référence d'un

autre type dont il a besoin, d'autre part tant qu'un langage ne comporte pas des "assertions sémantiques" (elles ont envisagées pour `java`), la phrase ci-dessus "avec la même signification reste incontrôlable.

2.2 Elements de langage, classes et objets.

La notion de base est celle de la classe, qui est en quelque sorte un gabarit de création d'objets. Ce gabarit est utilisé pour créer des instances d'objets. Un objet ayant en général besoin d'être initialisé, il est possible de définir des pseudo-méthodes d'initialisation, appelées usuellement constructeurs, qui sont automatiquement exécutées au moment de l'instantiation. Des méthodes seront ensuite invoquées sur les objets instantiés.

Outre les attributs et propriétés (nous parlerons de membres) décrits ci-dessus, qui sont associés à des instances spécifiques d'objet , une classe peut aussi posséder des membres qui ne sont pas liés à un objet particulier, appelés attributs ou méthodes de classe. Dans certains cas il est possible et intéressant de définir des classes qui ne sont même pas destinées à être intanciées, et contiennent un ensemble de "méthodes" qui ressemblent à des fonctions traditionnelles déguisées, mais néanmoins régroupées logiquement. Ces classes sont souvent désignées sous le nom de classes utilités.

Les langages objet possédent généralement une notion de protection, c-a-d que seuls certains membres d'une classe, dits publics, sont destinées à être exposées au code extérieur à la classe, les autres, dits privés, étant réservé à son propre usage, à des fins d'implémentation. On trouve parfois une graduation plus fine que les seuls niveaux publics et privés (c'est le cas de `java` et de `C++`), et la notion de protection s'applique souvent aussi aux classes elles-mêmes. Il est en général déconseillé d'exposer directement des attributs comme publics, souvent le monde extérieur n'a pas besoin d'être conscient de leur existence, et même s'il doit l'être, il est préférable de définir une ou deux méthodes, appelées "accesseurs", pour lire et/ou modifier leur valeur.

Si le langage est hybride, une notion plus traditionelle de fonction peut coexister avec les éléments purement objets.

2.3 La position de `java`

Comment se positionne `java` dans la gamme des langages orientés objet? Si l'on considére les deux langages les mieux connus:

- `C++` est un langage hybride (il y a de nombreux types primitifs, on peut écrire du code `C`, c'est à dire une sorte d'assembleur structuré!), et fortement typé.
- `SmallTalk` est une langage pur, non typé.

`java` occupe une position intermédiaire:

- Il est hybride, mais beaucoup moins que `C++`: il y a toujours des types primitifs (numériques+`char`+`void`), mais le développeur ne peut rien définir

qui soit en dehors d'une classe, toute classe définie par l'utilisateur dérive d'une classe prédéfinie (Objet).
- Il est aussi fortement typé que C++

Par ailleurs, java s'est appuyé sur une syntaxe très proche de celle du C++, après avoir éliminé les complexités les plus redoutables, au prix bien sûr de se priver de certaines possibilités.

2.4 Le Garbage Collector

Instantier un objet signifie le créer, ce qui pose le problème du cycle de vie et de sa destruction. Les langages objets épargnent en général la gestion de cette destruction au développeur. java est dans ce cas, contraitement à C++ qui, pour satisfaire les besoins en performance d'une partie de son marché, laisse au développeur le soin de spécifier allocation et déallocation. Le "Garbage Collection" a longtemps eu très mauvaise réputation, à cause de certaines implémentations de Lisp où l'exécution de cette phase provoquait des pauses *très* perceptibles dans l'exécution des applications. La banalisation du multithread, en permettant d'exécuter ce nettoyage en parallèle, a encore une fois dédramatisé le problème. Il reste que combiner Garbage Collection et Temps Réel est, sinon impossible, tout au moins délicat.

3 Les programmes java

3.1 Les environnements de développement

3.1.1 Le jdk

Le produit de base est le "Java Development Toolkit" de Sun. Il contient une implémentation de référence de la machine virtuelle, les librairies fondamentales nécessaires à son fonctionnement, un compilateur, et quelques outils supplémentaires. Il n'y a pas d'outil de développement visuel, bien que le " Bean Development Toolkit" contienne, lui, un tel outil, en la personne de la beanbox. Il n'y a malheureusement pas non plus, pour l'instant, de debugger digne de ce nom.

3.1.2 Les environnements intégrés

Beaucoup de constructeurs proposent des environnements intégrés avec debugger, développement visuel, etc... Les plus connus sont:

- Visual J++ de Microsoft
- JBuilder d'Imprise (ex Borland)
- VisualAge for Java d'IBM
- Visual Cafe de Symantec
- Kawa de Tek-Tools

3.2 L'architecture des programmes `java`

3.2.1 Structure logique, classes et packages

Si `java` comporte des types primitifs, il reste un langage objet "plutôt" pur, car le développeur ne peut définir que des classes. Le composant de base est donc la classe. Les classes peuvent, nous l'avons dit être regroupées en packages. les packages jouent deux rôles:

1. Ils représentent un espace de nommage, analogue au `namespace` de `C++`. Ceci évite entre autres les collisions de noms entre fournisseurs de composants.
2. Ils fournissent un niveau particulier de protection: il y a en `java`, comme en `C++`, 3 niveaux explicites de protection:

   - `private`: uniquement pour un membre, ou une classe intérieure, seule la classe englobante peut y faire référence
   - `protected`: similaire, mais l'accès est autorisée également aux classes dérivées de la classe englobante
   - `public`: membres ou classes, accès autorisé pour tous

   Mais contrairement à `C++`, en l'absence de qualificateur explicite, le défaut n'est pas un de ces 3 niveaux, mais un 4ème, le niveau "package", c-à-d accès autorisé à toutes les classes du même package.

Une classe est spécifiée comme appartenant à un package en préfixant son code par la syntaxe:

```
package test;
```

ou

```
package com.omg.Corba;
```

Comme on le voit dans ce dernier exemple, les packages peuvent être imbriqués. Une classe sans spécification de "`package`" est considérée appartenir à un package global au nom "vide". Les packages (non vides) font partie intégrante du nom de la classe, ainsi dans

```
package test;
public class Benchmark {...}
```

le nom complet de la classe est `test.Benchmark`. Une référence depuis le code d'une classe à une classe appartenant à un autre package doit normalement spécifier ce nom complet. On peut s'en dispenser avec la directive d' `import`: ainsi, dans

```
Package test;
import framework.*;
```

```
public class Benchmark {
  ...
  Application app=new Application();
  ...
}
```

`Application` peut faire référence à `framework.Application` (il est déconseillé que le package `test` contienne aussi une classe appelée `Application`!). On aurait pu obtenir le même résultat, de façon plus économique, et plus sûre, avec:

```
Package test;
import framework.Application;
public class Benchmark {
  ...
  Application app=new Application();
  ...
}
```

3.2.2 Structure physique, les fichiers

L'unité de pseudo-code sera donc le code compilé définissant totalement une classe. Le nom du fichier est le nom exact de la classe suivi du suffixe "`.class`". L'architecture des sources est presque aussi simple: un source java doit contenir au plus la définition d'une classe publique, et le nom du fichier doit alors être le nom exact de cette classe suivi du suffixe " `.java`". Une classe peut avoir besoin de classes "auxiliaires" privées, celles-ci peuvent être définies dans le même fichier source, même si elles donneront lieu à une unité de code compilé indépendante. D'autre part, depuis jdk1.1, on peut définir des classes "intérieures" à une autre classe. Pour le code compilé d'une classe appartenant à un package, les segments du nom du package sont traduits non pas directement dans le nom de fichier mais par des sous-répertoires successifs. Ainsi le code compilé de la classe `test.Benchmark` devra se trouver dans un fichier dont la spécification sera de la forme:

xxx`\test\Benchmark.class`

Bien qu'il soit souhaitable de respecter une cohérence identique pour les sources, aucune contrainte n'est imposée sur celles-ci, ni par la spécification, ni par les outils.

3.2.3 La compilation

Dans le cas du jdk, la compilation est exécutée en lançant depuis la ligne de commande l'utilitaire `javac`. La syntaxe est:

`javac` *[options] fichier*`.java`

Les options les plus utilisées sont:

- `-classpath` suivi d'une liste de répertoires où rechercher les classes référencées par le source compilé, séparés par des " ;" sous Windows ou par des ":" sous Unix.
- `-d` suivi de la spécification du répertoire où écrire le code compilé

Si la classe appartient à un package, l'argument de `-d` sera la racine relative du répertoire effectif, c-a-d que:

```
javac -d C:\dev\classes benchmark.java
```

produira (en supposant que nous soyons dans le répertoire où se trouve le source de la classe `test.Benchmark`) le fichier:

```
C:\dev\classes\test\Benchmark.class
```

`classpath` est interprété de la même façon, et si `Benchmark` référence la classe `framework.Application` dont le compilé se trouve dans:

```
C:\dev\classes\framework\Application.class
```

l'argument de `-classpath` devra contenir, entre autres, `C:\dev\classes`

3.2.4 L'exécution et les machines virtuelles

Le code compilé est exécuté par la machine virtuelle. Quand une nouvelle classe est requise (par exemple, parce qu'elle est référencée par une classe déjà chargée. Nous verrons plus loin comment la première classe est chargée), la machine virtuelle charge le code en mémoire puis exécute des opérations regroupées en 3 phases:

1. Vérification:
   - Contrôler que le bloc chargé commence bien par le mot magique qui identifie les classes `java` compilées
   - Contrôler que le code compilé ne comporte que des "bytecode" légaux
   - Contrôler que tout branchement débouche bien sur un début d'instruction
   - bien d'autres choses...
2. Préparation: principalement création des variables de classes et leur affectation avec les valeurs par défaut ou des valeurs initiales spécifiées tant qu'elles sont déterminées à la compilation
3. Résolution: localiser les classes référencées symboliquement et les charger. Cette phase peut-être retardée jusqu'au moment où une classe donnée est absolument nécessaire.

Quand la machine virtuelle est lancée explicitement, par exemple depuis la ligne de commande, elle accepte une option `-classpath` identique à celle du compilateur.

3.2.5 Applications, applets, servlets

Une application autonome doit posséder un point d'entrée standard, sous la forme d'une classe publique, possédant une méthode publique de classe dont le nom est `main`. Plus précisément l'en-tête de cette méthode (sa signature) doit être:

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

`static` signifie que c'est une méthode de classe et pas une méthode d'instance, `void` signale qu'elle ne renvoie aucun résultat, et `(String[] arg)` qu'elle reçoit comme un argument un tableau de chaînes de caractères. Si une application est lancée par:

```
java FileCopy input output
```

`FileCopy.main` recevra comme argument un tableau de 2 chaînes de caractères: {"input","output"}.

Une telle application est exécutée en lançant l'exécutable de la machine virtuelle avec le nom de la classe en argument. Certains environnements d'exécution fournissent une application cadre (framework), qui gère des composants, qui n'ont donc pas besoin d'un `main()` propre, mais dont la responsibilité est d'exporter d'autres méthodes. Les deux cas les plus fréquemment rencontrés sont:

- L'environnement d'exécution d'applets au sein d'un navigateur web. Les applets sont des classes `java` qui enrichissent le contenu de pages web, et qui doivent posséder certaines méthodes d'initialisation, de lancement, de suspension et de terminaison. Ce sont normalement des composants graphiques qui doivent donc réagir à des évènements, et se représenter dans la page. L' exécution d'un applet est déclenchée par le chargement dans le navigateur d'une page html contenant un référence à cet applet.
- l'environnement d'exécution de servlets au sein d'un serveur web. Les servlets sont des classes java qui permettent de générer, en réponse à des requêtes en provenance d'un navigateur lointain, des pages web dynamiques, et qui doivent posséder des méthodes de traitement de requêtes HTTP. L'exécution d'un servlet est déclenchée par la réception par le serveur d'une requête dont l'url référence ce servlet.

3.3 Les packages de base

Un certain nombre de packages font partie de la spécification du langage. La liste suivante n'est pas exhaustive:

- Les packages noyau: bien qu'ils soient séparés pour des raisons logiques, chacun de ces packages fait référence aux autres, et ne pourrait fonctionner sans eux, même si la spécification `java` l'autorisait:
  - `java.lang` C'est le seul package qui n'ait jamais besoin d'être importé. Il contient entre autres la class `Object`, les classes `String` et

`StringBuffer` déjà rencontrées, la classe `Exception`. Il contient aussi la classe utilitaire `System`, celle-ci est une classe "utilité", ne contenant que des membres de classe, et en particulier s'appuie sur le package `java.io`, par exemple pour fournir l'équivalent de stdout et stderr.

  - `java.util` Un package d'intérêt très général, mais contenant des classes moins "système": la très utilisée classe `Vector` se trouve ici. Ce ne sont *pas* des classes "utilités", `util` signifiant ici "utilitaire", ce qui posséde (malheureusement) un sens différent sous une apparence proche!
  - `java.io` Le package de base d'accès aux entrèes-sorties (fichiers, terminaux).

- `java.awt` Le package initial pour la programmation des "GUI". Un des points les plus remaniés de `java` depuis sa sortie (à juste titre!), il est maintenant conseillé d'utiliser `javax.swing`.
- `java.applet` Le package de définition des applets.
- `java.net` Le package pour la programmation de réseau.
- `java.beans` le package des composants `java`.
- `java.security` Le package de la sécurité: signature, encryption, politique de sécurité. En fait bien que ce package puisse être ignoré dans un premier temps par le développeur applicatif, c'est absolument un package "noyau", et les 3 premiers packages y font référence.

En principe, les packages dont la racine est " `java`" sont ceux qui font partie des spécifications du langage, et ceux dont la racne est " `javax`" sont des extensions. Néanmoins `javax.swing` fait partie intégrante de jdk1.2, et son inclusion dans `javax` est basée sur ce que l'on appelle en général pudiquement des "raisons socio-historiques".

4 Eléments de base du langage

4.1 Eléments lexicaux

Le jeu de caractères utilisé pour l'écriture des programmes est l'Unicode, qui étend les codes ASCII pour y inclure tous les caractères nationaux de tous les pays. Les caractères accentués sont typiquement acceptés comme "lettres" dans la définition des identificateurs. Ceci signifie qu'un développeur Français qui le désire peut déclarer des variables comme "résultat". En théorie ceci est aussi vrai pour les noms de classe, mais ces noms ayant des conséquences sur les noms de fichier, les environnements et les outils peuvent mal réagir à de tels noms, et la prudence est encore conseillée.

4.2 Types, valeurs, et variables

`java` est un langage fortement typé, ce qui veut dire que toute variable doit être déclarée et typée avant qu'on puisse y faire référence, et que toute expression

posséde un type déterminable à la compilation.
Les types rentrent dans trois catégories:

1. Les types primitifs:
   - Les types intégraux signés:
     - `byte` sur 8 bits (attention, en `C/C++`, `byte` est souvent *`unsigned`* `char`)
     - `short` sur 16 bits
     - `int` sur 32 bits
     - `long` sur 64 bits

     Le modificateur `unsigned` n'existe pas en `java`. Les valeurs de ces types, et leurs constantes littérales, sont celles du `C++` sur les machines où les tailles coïncident. Le fait que la taille de ces types soit justement , en `java`, indépendante de l'environnement, s'il peut être un blocage rhédibitoire pour les développeurs qui *doivent* coller à la machine, est un immense soulagement pour tous les autres!
   - `char` un entier non signé sur 16 bits, représentant un code Unicode
     Les valeurs et constantes littérales sont, cette fois, un sur-ensemble du `C++`, l'Unicode ajoutant les séquences:

     `'\u`*`xxxx`*`'`

     où *`xxxx`* représente 4 chiffrs hexadécimaux.
   - Les types flottants IEEE signés
     - float sur 32 bits
     - double sur 64 bits
   - `boolean` représentant une "valeur de vérité", et dont les valeurs possibles sont `true` ou `false`
   - `void` qui est un type sans valeur!

2. Les types qui référencent une classe.
   Si l'on a une déclaration:

   ```
   Car c0;
   ```

   en supposant que Car soit une classe définie dans un package courant ou importé, sa signification est que l'identificateur o1 est une variable qui dénote un objet de type Car. L'instruction:

   ```
   c0=c1;
   ```

   ne signifie pas que `c0` est modifié en quoi que ce soit, ni qu'une copie de `c1` est créée, mais que `c0` dénote maintenant le même objet que `c1`. Si `c0` est modifiable, ce sera par:
   - `c0.`*`attribute=value`*`;` en admettant que `Car` posséde au moins un attribut de classe modifiable

– `c0.`*`modifyingMethod`*`();` en admettant que `Car` possède au moins une méthode modifiante

Si `c0` est passé en argument à une méthode:

```
public void use(Car car) {...}
```

par l'instruction

```
xxx.use(c0);
```

ce n'est pas une copie qui est transmise, mais bien une référence partagée sur l'objet initial, et si la méthode appelée modifie cet objet ci-dessus, ces modifications seront visibles à travers `c0`. Par contre, si la méthode exécute

```
car=anotherCar;
```

`c0` n'est pas touché. Noter que, les types primitifs ne possédant ni méthodes ni attributs, ils ne peuvent être modifiés en étant passés comme argument. On dit quelquefois que les valeurs primitves sont passées par valeur, alors que les objets sont passés par référence. En fait, en `java`, *tous les arguments* sont *toujours* passés *par valeur*, mais les valeurs qui dénotent des objets sont elles-même des références!
Nous verrons d'autre part, avec l'héritage, que dans le cas de la déclaration "`Car c0;`", à l'exécution, `c0` peut référencer un objet d'un type "plus riche" que `Car`.

3. Les types obtenus à partir d'un autre type par l'adjonction de "`[]`", c'est-à-dire les types tableaux.
En réalité un type tableau est une classe, que son type de base soit primitif ou non, généré automatiquement par le système compilateur/machine virtuelle. Le type ne spécifie pas la dimension d'un tableau, mais un tableau existant ne peut modifier cette dimension: il possède un attribut `length` qui se comporte exactement comme une variable membre d'instance déclarée `final`. Les éléments individuels du tableau sont accédés par la syntaxe:

```
array[i]
```

la machine virtuelle vérifiant que la valeur de `i` est compatible avec la dimension du tableau, et ils peuvent être modifiés:

```
array[i]=new_element;
```

Les tableaux multi-dimensionnels sont réalisés comme des tableaux de tableaux.

Le type "chaîne de caractères" n'apparaît pas explicitement: en effet ce n'est pas un type primitif, mais une classe prédéfinie, `String`, à laquelle le compilateur réserve un traitement spécial: en effet les constantes littérales de ce type ont la même forme qu'en `C/C++` ("`Hello World!`", "`c:\\winnt`"), l'opérateur `+` est surchargé de son sens usuel numérique pour exprimer la concaténation, tous les autres types sont susceptibles d'une "conversion" en `String` qui peut être déclenchée implicitement, par exemple quand une expression de ce type est un opérande de la concaténation. Un objet de type `String` ne peut jamais être modifiée: en effet les seuls membres d'instance non privés sont des méthodes non modifiantes. Les méthodes dont le nom semble "modifiant" génère en fait de nouvelles chaînes Dans le code suivant:

```
String unixPath= expression ;
String winPath=unixPath.replace('\\','/');
```

`unixPath` est inchangé. Pour avoir l'équivalent d'un tableau de caractères modifiables, on doit utiliser la classe `StringBuffer`, qui est l'équivalent d'un `char[]` dont la dimension serait modifiable. On peut convertir aisément un `StringBuffer` en `String` et vice-versa.

4.3 Les déclarations

4.3.1 Les variables

Les déclarations de variable se trouvent aussi bien dans le corps qu'une méthode que directement à l'intérieur d'une classe.Une déclaration est toujours de la forme:

```
type identificateur [= valeur_initiale] ;
```

ce qui est beaucoup plus simple que la syntaxe correspondante en `C/C++`. L'initialisation est optionelle, son absence impose certaines contraintes. Une déclaration peut concerner:

1. Une variable membre
   (a) d'instance
   (b) de classe, la déclaration ci-dessus est alors précédée du mot-clef `static`
2. Une variable locale

Les conséquences de l'absence d'initialisateur dans la déclaration varient suivant les cas. Dans les cas 1, une valeur par défaut est affectée. Cette valeur est

- Pour les types numériques, le "`0`" du type concerné (par exemple, `'\u0000'` pour `char`)
- Pour un `boolean`, `false`
- Pour un type "référénce sur un object", y compris un type tableau, `null`, qui signifie que la variable ne réfèrence aucun objet

Dans le cas 2, aucune valeur par défaut n'est affectée, mais le compilateur exige de pouvoir se convaincre que la variable recevra une valeur avant toute utilisation.
Une variable peut aussi être qualifiée de `final`, ce qui signifie qu'elle ne doit prendre qu'une seule valeur au cours de son cycle de vie. Là plus encore, en l'absence d'initialisation, les exigences du système sont variables:

- Dans le cas 2, le mécanisme est le même que pour les variables non `final`, mais bien sûr en outre le compilateur refuse toute autre affectation
- Dans le cas 1(b), une initialisation est exigée (mais le message d'erreur de `javac` est un peu confus)
- Dans le cas 1(a), les valeurs par défaut ne sont pas utilisées, *tout constructeur* doit comporter une affectation à ce membre, bien sûr avant tout usage, et non suivie d'une autre.

Du point de vue du style, c'est-à-dire de la lisibilité et de la maintenance du code, il est toujours avantageux si cela est possible de ne déclarer une variable locale que quand elle peut recevoir une initialisation significative. Dans certains cas, néanmoins, l'initialisation doit être réalisée dans des branches conditionnelles, et le compilateur garantit que le développeur ne pourra pas "oublier" un cas. Le traitement des variables `final` membres d'instance est particulièrement intéressant, car il permet de déclarer des attributs non modifiables dont la valeur dépend néanmoins des arguments du constructeur.

4.3.2 Les méthodes

La déclaration d'une méthode prend la forme:

```
return-type methodName(argument-list) optional-throws-clause {
  methodBody
}
```

où:

`return-type` est ou bien le type de l'expression renvoyée par l'exécution de la méthode, ou bien `void`.

`argument-list` est une liste de déclarations sans qualificateurs ni initialisateurs séparée par des virgules, qui décrit les arguments attendus par la méthode. Cette liste peut être vide.

`optional-throws-clause` a, si elle est présente, la forme `throws exception-list`, nous reviendrons plus tard sur les exceptions et cette clause.

`methodBody` est une liste d'instructions (voir la définition d'une instruction plus loin) qui ne peut être vide que si *`return-type`* est identique à `void`, sinon tout chemin d'exécution doit se terminer par une instruction `return` renvoyant une expression dont le type est compatible avec *`return-type`*.

4.4 Les expressions

4.4.1 Structure des expressions

La syntaxe des expressions `java`, et, jusqu'à un certain point, leur signification, est très voisine du `C++`. Une expression est composée à partir des éléments suivants:

1. Les constantes littérales. Il en existe de type

   (a) `int`: `0`, `-100`, `1234567890`
   (b) `long`: `0L`, `-100L`, `1234567891234567890L`
   (c) `char`: `'a'`, `'\n'`, `'\\'`, `'\"'`, `'é'`, `'\u03e9'`
   (d) `float`: `0f`, `3.14f`, `-6.022e+15f`
   (e) `double`: `0.`, `3.14`, `1e137`
   (f) `boolean`: `false` et `true`
   (g) `String`, bien que ce ne soit pas un type primitif: "", "`Hello`", "`One\nTwo`". En outre le compilateur transformera la concaténation de deux constantes littérales de type `String`:

   ```
   String msg="Hello "+"World!";
   ```

   sera compilé exactement comme

   ```
   String msg="Hello World!";
   ```

2. Les variables. Ce sont des identificateurs reconnus comme définis et valides dans le bloc d'instructions courant en tant que (certains cas font référence à des notions décrites plus loin):

   (a) Une variable locale, introduite par une définition normale ou
   (b) Une variable introduite par l'initialisation d'une instruction `for` en cours d'exécution.
   (c) Un argument de la méthode ou du constructeur en cours d'exécution.
   (d) L'identificateur lié à l'exception en cours de traitement si un bloc de traitement d'exception est en cours d'exécution.
   (e) Une variable membre d'instance de `this`, si la méthode en cours d'exécution est une méthode d'instance ou un constructeur.
   (f) Une variable membre de classe de la classe dont une méthode (d'instance ou de classe) ou un constructeur est en cours d'exécution.

   Les cas (a) à (d) sont en fait des variantes de variables locales, c'est-à-dire des variables dont la durée de vie n'est pa s liée à celle d'une instance ou d'une classe, mais à la durée d'exécution d'un bloc de code. En `java`, une variable locale ne peut *jamais* en cacher une autre, c'est-à-dire redéfinir une autre variable locale de même nom qui serait visible au point de la nouvelle définition.

3. Les opérateurs :
   (a) unaires: `++`, `--`, `+`, `-`, `~`, `!`
   Comme en `C/C++`, `++` et `--` existent en préfixé et en postfixé
   (b) binaires, groupés par précédence:
      i. multiplicatifs: `*`, `/`, `%`
      ii. additifs: `+`, `-`
      iii. de shift: `<<`, `>>`, `<<<`, `>>>`
      iv. de comparaison: `<`, `<=`, `>`, `>=`
      v. d'égalité: `==`, `!=`
      vi. de manipulation binaire: `&`, `|`, `^`
      vii. logiques: `&&`, `||`,
   (c) ternaire: *cond* `?` *expr0* `:` *expr1*
4. Les opérations liées au "transtypage"
   (a) Les conversions: `(`*type*`)` *expr*
   (b) La comparaison de type: *expr* `instanceof` *referenceType*
5. Les affectations: `=`, et *op*`=`, *op* étant un des opérateurs binaires définis en (b)i, (b)ii, (b)iii, ou (b)vi.
6. L'opérateur d'accès à un élément d'un tableau, concrétisé par " `[]`": *expr*`[`*index*`]`
7. L'opérateur d'accès à un attribut, concrétisé par " `.`":
   (a) *expr*`.`*attrName*
   (b) *referenceType*`.`*attrName*
8. Les invocations de méthodes:
   (a) *expr*`.`*methName*`(`*arguments*`)`
   (b) *referenceType*`.`*methName*`(`*arguments*`)`
9. Les instantiations d'objet: `new` *class*`(`*arguments*`)`

En dehors de leurs utilisations syntaxiques explicites (conversions, déclarations et invocations de méthode, instructions de boucle, traitements d'exceptions) les parenthèses sont utilisées pour grouper des sous-expressions, soit par lisibilité, soit pour imposer une signification différente de celle impliquée par les précédences par défaut. Par exemple:

```
a*x+y
```

a la même signification que

```
(a*x)+y
```

et non pas la même que

```
a+(x*y)
```

Contrairement à `C/C++`, "`,`" n'est *jamais* un opérateur, il intervient uniquement comme séparateur, dans les cas suivants (certains font référence à des notions définies dans des sections ultérieures):

- entre plusieurs variables déclarées simultanément
- entre les paramètres formels d'une déclaration de méthode ou de constructeur
- entre les argments passés lors d'une invocation de méthode ou d'une instantiation d'objet
- entre plusieurs interfaces hérités par un même interface ou une même classe
- entre plusieurs exceptions déclarées levées dans une déclaration de méthode
- entre plusieurs instructions-expressions dans le code d'initialisation ou d'itération d'une instruction `for`.

4.4.2 Remarques sur les expressions

1. L'affectation exige que le membre gauche soit ce qu'on appelle an anblais une "lvalue", c'est-à-dire une expression dénotant non pas simplement une valeur, mais une "location" où une valeur peut être conservée. Les lvalue en java sont:

   (a) les variables, locales, ou membres de classe ou d'instances.
   (b) Une expression d'accès à un attribut de classe.
   (c) Une expression d'accès à un attribut d'instance dont le membre gauche est une lvalue.
   (d) Une expression d'accès à un élément de tableau dont le membre gauche est une lvalue.

   En outre, pour être affectable:

   (a) une lvalue ne doit pas être une variable `final` ayant déjà été initialisée.
   (b) le type du membre droit doit être compatible avec celui du membre gauche (c'est-à-dire être susceptible d'une conversion implicite dans celui-ci, voir plus loin des remarques additionnelles sur ce que signifie la compatibilité pour les types primitifs)
   (c) le membre gauche doit aussi satisfaire à toutes les conditions d'accessibilité, mais ceci n'est pas une condition liée à l'opérateur d'affectation, mais intrinséque à toute expression.

2. Les opérateurs `++` et `--` sont à considérer comme des opérateurs d'affectation.
3. Nous avons décrit plus haut quelles étaient les conversions explicites et implicites valides entre types référence. Bien que les types primitifs se situent en dehors de l'orientation objet, d'une certaine façon le type `int` est une extension du type `short`. Un type numérique sera généralement convertible implicitement en un type "plus étendu", une conversion explicite étant nécessaire dans le sens inverse. Il n'y a aucune conversion entre les types numériques et le type `boolean` (pas plus que `null` ne peut être converti en `false`). Attention: contrairement à ce qu'on pourrait penser, il existe des cas où une conversion implicite peut amener une perte silencieuse de précision:

   ```
   int i=1234567890;
   float f=i;
   ```

   Un `float` n'a pas suffisamment de précision pour conserver tous les chiffres significatifs de `i`. Le même problème se pose en affectant un `long` à un `double`.
4. Une conséquence de l'application stricte des régles qui précèdent est que la dernière instruction du bloc suivant génère une erreur de compilation:

   ```
   short s=2;
   int i=3;
   short t=i*s;
   ```

   En effet, la division est comprise comme "`i*(int)s`", puisqu'il s'agit d'une conversion implicite acceptable, et l'instruction cherche alors à affecter un `int` à un `short`, ce qui exige une conversion explicite. Des formes acceptées, ayant un sens différent, mais, dans ce cas, le même résultat, sont:

   ```
   short t=(short)(i*s);
   ```

   ou

   ```
   short t=(short)i*s;
   ```

   Le compilateur admet une exception à cetter régle pour les affectations de constantes calculables à la compilation, ce qui est bien agréable car il n'y a pas de constantes lexicales entières en dessous de l' `int`. Par exemple:

   ```
   short s=1;
   ```

   est légal, sans nécessiter "`(short)1`". Cette exception ne concerne *que* les affectations: si une méthode attend un argument de type `short`, il faudra lui passer "`(short)1`".

5. Comme nous l'avons déjà vu, les conversions implicites des type primitifs *vers* le type `String` sont une exception à ces régles. Elles sont en particulier déclenchées par l'opérateur binaire `+` considéré comme opérateur de concaténation, ce qui est déjà une interprétation particulière du compilateur.
6. Si les opérateurs de divison entière peuvent lever une `ArithmeticException` en cas de division par `0`, ce n'est pas le cas des opérations flottantes, qui utilisent les notions IEEE d'infinité et de NAN (NotANumber), et ne provoquent jamais d'exception.

4.5 Les instructions

Une fois encore, la syntaxe s'appuie sur le `C/C++`. Elles peuvent être regroupées en deux catégories, les instructions composées, qui peuvent contenir d'autres instructions , et les instructions simples, qui ne peuvent pas. Par ailleurs, toute instruction peut optionellement être précédée d'une étiquette, sous la forme:

identificateur :
instruction

le séparateur de ligne n'étant pas obligatoire.

4.5.1 Instruction simples

Les instructions simples doivent toutes être terminées par un " `;`", que nous ne répéterons pas dans leur description. Ce sont:

1. L'instruction vide: sans commentaires.
2. L'instruction-expression: l'expression est évaluée pour son "effet de bord", ce sera une affectation (y compris l'évaluation des opérateurs `++` et `--`), une invocation de méthode, ou une instantiation d'objet. Le compilateur rejetera toute autre expression.
3. L'instruction de retour de méthode. Deux formes:

   (a) `return` si la méthode n'a pas de type de retour
   (b) `return expr` si la méthode a un type de retour (autre que `void`) déclaré.

4. L'instruction `break`: "sortie" d'une instruction composée. Le mot-clef `break` peut optionnellement être suivi d'un identificateur qui doit être celui d'une étiquette d'une instruction composée englobant l'instruction courante. Si cette étiquette est spécifiée, elle identifie l'instruction "cible". Sinon, l'instruction "cible" est l'instruction composée de type boucle (`for`, `while`, ou `do`) ou `switch` (voir plus loin leurs descriptions) la plus intérieure englobant l'instruction courante. Si aucune instruction cible n'est identifiée le compilateur rejetera le `break`. Sinon l'effet est de transférer le contrôle

d'exécution à l'instruction suivant immédiatement l'instruction cible (ce peut être une instruction `return` implicite à la fin d'une méthode sans type de retour).

5. L'instruction `continue`: "enchaînement" de boucle. Le mot-clef `continue` peut optionnellement être suivi d'un identificateur qui doit être celui d'une étiquette d'une instruction composée englobant l'instruction courante. Si cette étiquette est spécifiée *et si l'instruction identifiée est une instruction de boucle*, elle identifie l'instruction "cible". Sinon, l'instruction "cible" est l'instruction composée de type boucle la plus intérieure englobant l'instruction courante. Si aucune instruction cible n'est identifiée le compilateur rejetera le `continue`. Sinon l'effet est de transférer le contrôle d'exécution au point de l'instruction de boucle qui enchaîne sur l'itération suivante.
6. L'instruction `throw`: sa syntaxe est:

   `throw` *`exceptionExpr`*

   `exceptionExpr` devant s'évaluer comme une instance dérivée de `Throwable` (normalement `Exception`), et son effet est de lever l'exception en question, et de transférer "abruptement" le contrôle d'exécution au premier point de code appelant ayant déclaré traiter une exception de ce type (voir plus loin l'instruction `try`).

4.5.2 Instructions composées

1. Le bloc: la syntaxe du bloc est une liste d'instructions encadrées par " `{`" et "`}`". Cette liste peut être vide, "`{}`" est donc une instruction composée légale de type bloc.
2. Le bloc synchronisé: la syntaxe est celle du bloc précédée de " `synchronized(` *`expr`* `)`". `expr` doit s'évaluer comme une référence non `null` à un objet. Si `expr` possède un type primitif, le compilateur rejetera l'expression. Sinon, et si `expr` s'évalue comme `null`, une exception sera levée à l'exécution
3. L'instruction conditionnelle: sa syntaxe est:

   `if (` *`booleanExpr`* `)` *`yesStatement`* `[else` *`noStatement`*`]`

   `yesStatement` et *`noStatement`* étant des instructions quelconques, et la partie `else` étant optionnelle.
4. L'instruction de branchement: sa syntaxe consiste en l'encadrement par "`switch(expr) {`" et "`}`" d'une suite d'"étiquettes de choix" et d'instructions, une "étiquette de choix" étant soit:

   `default:`

   soit:

   `case` *`constantExpr`* `:`

Cette forme est soumise à certaines contraintes:

- *expr* doit être d'un type intégral autre que `long`.
- les *constantExpr* doivent être déterminées au moment de la compilation et affectables au type de *expr*.
- deux étiquettes de choix dans la même instruction de branchement ne peuvent pas être identiques

A l'exécution de cette instruction, *expr* est évaluée et une des étiquettes est choisie en fonction de sa valeur . Si cette valeur ne corrrespond à aucun des "`case ...:`" et si l'étiquette `default` n'est pas présente, le contrôle est transféré à l'instruction suivant l'instruction de branchement. Sinon le contrôle est transféré à l'instruction précédée par l'étiquette. Cette instruction est certes utile, mais avant même de ne pas être orientée objet, elle est aussi peu compatible avec la programmation structurée. C'est en fait un héritage quasi-direct de l'assembleur!

5. L'instruction de traitement d'exception. Sa syntaxe est:

   `try` *tryBlock*
   `catch(`*ExceptionClass id*`)` *handlerBlock*
   ...
   [`finally` *finalBlock]*

   block représentant une instruction de type bloc, les "`...`" signifiant que la clause `catch` peut être répétée, et les "*[]*" signifiant que la clause `finally` est optionelle (en réalité un tel bloc peut aussi comporter une clause `finally` sans aucune clause `catch`). Les *ExceptionClass* doivent être des types références dérivés de `Throwable`. A l'exécution:

   (a) *tryBlock* est exécuté.
   (b) Si une exception est levée et non traitée à l'intérieur de cette exécution, et si son type est compatible avec l'*ExceptionClass* d'une ou plusieurs clauses `catch`:
      i. le contrôle est transféré au *handlerBlock* de la première telle clause, *id* référençant l'objet exception qui a été levé.
      ii. si l'exécution de ce bloc lève une nouvelle exception, elle est propagée vers les niveaux supérieurs
   (c) Si aucune clause ne satisfait la condition précédente, l'exception originelle est propagée vers les niveaux supérieurs.
   (d) Si une clause `finally` est présente, dans tous les cas *:*
      i. *finalBlock* est exécuté après terminaison du *mainBlock* et/ou du *handlerBlock*
      ii. si cette exécution lève une exception, elle est propagée vers les niveaux supérieurs, remplaçant éventuellement l'expression mentionnée en (b)ii ou (c).

6. L'instruction `for`: la syntaxe en est:

   `for (`*`[init]`*`;`*`[booleanExpr]`*`;`*`[next]`*`)` *`statement`*

   `statement` étant une instruction, les " *`[]`* " indiquant un caractère optionnel, *`init`* étant soit une déclaration de variable locale avec initialisation, soit une liste d'instructions-expressions séparées par des virgules, *`next`* étant une liste d'instructions-expressions séparées par des virgules. L'exécution de l'instruction `for` se décompose ainsi:

   (a) Initialisation:

      i. Si *`init`* est une déclaration de variable, elle est exécutée, la portée de la variable étant le reste des composants du `for`.
      ii. Sinon, la ou les instructions composant *`init`* sont exécutées dans l'ordre.

   (b) Itération:

      i. Si *`booleanExpr`* est présent et si son évaluation renvoie `false`, l'instruction `for` est terminée
      ii. Sinon, *`statement`* est exécuté, la ou les instructions composant *`next`* sont exécutées dans l'ordre, et une autre itération est entamée.

7. L'instruction `while`: la syntaxe en est:

   `while (`*`booleanExpr`*`)` *`statement`*

   L'exécution ne comporte qu'une phase, l'itération:

   (a) Si l'évaluation de *`booleanExpr`* renvoie `false`, l'instruction `while` est terminée
   (b) Sinon, *`statement`* est exécuté, et une autre itération est entamée.

8. L'instruction `do`: la syntaxe en est:

   `do` *`statement`* `while (`*`booleanExpr`*`);`

   ce qui en fait la seule instruction composée à se terminer par un " ;". L'exécution ne comporte qu'une phase, l'itération:

   (a) *`statement`* est exécuté
   (b) Si l'évaluation de *`booleanExpr`* renvoie `false`, l'instruction `do` est terminée, sinon une autre itération est entamée

5 Les classes de java

5.1 Définition de base

Au minimum, la définition d'une classe prend la forme:

```
class ClassName {
  // déclaration des membres
}
```

les membres étant des variables ou des méthodes, d'instance ou de classe, et des constructeurs. La déclaration d'un membre de classe est identifiée en étant précédée du qualificateur `static` (Depuis `jdk1.1`, un membre peut également être une "classe intérieure"). Nous savons déjà que la classe peut être optionellement déclarée `public`. La déclaration de chaque membre peut aussi être précédée d'un qualificateur de protection, `public`, `protected`, ou `private`.

5.2 Attributs

La déclaration d'une variable membre est identique à la déclaration d'une variable locale. En dehors du qualificateur `final`, applicable également aux variables locales, du qualificateur `static`, applicable également à tous les membres à l'exclusion des constructeurs, et des qualificateurs de protection, applicables également à tous les membres, deux qualificateurs sont réservés aux attributs:

- `transient`, lié à la sérialisation
- `volatile`, lié à la concurrencr

5.3 Méthodes

Les méthodes sont, en `java`, la seule forme des fonctions. On a déjà vu qu'on ne peut *rien* définir en dehors d'une classe, mais on ne peut pas non plus définir de fonctions locales au corps d'une méthode. Si l'on a besoin de fonctions n'étant pas associées à un objet possédant un état (c'est-à-dire un ensemble d'attributs) propre, ces fonctions seront définies comme méthodes de classe d'une classe correspondant à leur fonctionnalité.

Plusieurs méthodes d'une même classe peuvent avoir le même nom si le nombre ou les types de leurs arguments sont différents. *La seule différence des types de retour n'est pas suffisante!* Dans le code d'une méthode d'instance, la pseudo-variable `this` est automatiquement définie par le compilateur, et dénote l'objet sur lequel la méthode a été invoquée.

5.4 Constructeurs

Un constructeur est une pseudo-méthode n'ayant pas de spécification de type de retour, et dont le nom est celui de la classe. Bien que le constructeur soit

plus proche d'une méthode de classe que d'une méthode d'instance, la pseudo-variable `this` est définie et dénote l'objet qui est en cours d'instantiation. Une classe peut posséder plusieurs constructeurs, toujours différenciés par leurs listes d'arguments. Si aucun constructeur n'est déclaré, un constructeur implicite, public, sans arguments, et avec un corps vide, est généré.Un constructeur peut en appeler un autre par une utilisation patriculière de `this`:

```
public class Container {
  public Container(int size) {...}
  public Container() {
    this(10);
  }
}
```

`this(10)` est ici interprété comme appelant le constructeur approprié à la liste d'arguments qui lui est passé. L'exemple ci-dessus permet de réaliser une notion de valeur par défaut de l'argument `size`. Attention:

- Un appel à `this` avec des arguments de même type que le constructeur courant provoquera une récursion infinie. heureusement les compilateurs le détectent.
- Un constructeur qui en appelle un autre peut exécuter du code supplémentaire, mais l'appel à `this` doit être la première instruction.

L'appel du constructeur est déclenché par la syntaxe:

```
var=new ClassName(args);
```

la liste d'arguments pouvant bien sûr être vide. Cette syntaxe est la seule pouvant déclencher la création d'un objet. Dans le cas des tableaux, que nou savons être des classes déguisées, le constructeur est accessible par une syntaxe particulière:

```
String[] names=new String[n];
```

qui alloue un tableau de n chaînes, dont tous les éléments (traités comme des attributs de la classe `String[]`) sont initialisés à `null`. Dans le cas des tableaux multi-dimensionnels, cette syntaxe peut s'étendre sous deux formes:

```
String[][] names=new String[p][q];
```

qui alloue un tableau de `p` tableaux de `q` chaînes, chacun des `p*q` éléments initialisés à `null`, ou

```
String[][] names=new String[p][];
```

qui alloue un tableau de `p` tableaux de chaînes, chacun de ces `p` élements de type `String[]` étant lui-même initialisé à `null`.

5.5 Destruction

Comme on l'a dit, la destruction d'un objet n'est jamais demandée explicitement par le programme. Un objet "vit" tant qu'il est référencé, les références pouvant provenir directement:

1. d'une variable locale
2. d'une variable membre de classe d'une classe chargée
3. d'une variable membre d'instance d'un objet "vivant"
4. d'un élément d'un tableau, ce qui est une variante particulière du cas précédent, et très répandue: tous les "containers" existant couramment utilisent en définitive un tableau.

Dès qu'un objet n'est plus référencé, la machine virtuelle est libre de libérer son espace mémoire, grâce au "Garbage Collector" mentionné plus haut. Néanmoins, l'initialisation de l'objet peut avoir alloué des ressources précieuses (comme des connexions réseau) dont la machine virtuelle n'est pas consciente. Une méthode particuliére:

```
void finalize() {...}
```

est définissable comme l'endroit adéquat pour libérer les ressources autres que la mémoire allouées par l'objet. Son rôle est similaire à celui du destructeur en `C++`. Néanmoins, si la machine virtuelle est tenue de détruire les objets non référencés avant de se plaindre qu'il n'y a plus de mémoire pour en créer de nouveaux, et, si possible avant que l'utilisateur ne se plaigne que l'application rame (!), elle est, dans ces limites, totalement libre du moment de ces destructions, et de leur ordonnancement. Ceci, joint au fait que le développeur ne peut pas choisir entre allocation dynamique des objets et allocation automatique sur la pile, fait que le rôle du finaliseur est beaucoup moins crucial et prépondérant qu'en `C++`.

5.6 Héritage

Une classe peut hériter d'une autre. La syntaxe pour ceci est:

```
class DerivedClass extends BaseClass {...}
```

Une classe ne peut "étendre" qu'une seule classe de base, ce qui élimine l'héritage multiple que l'on trouve dans d'autres langages objet. Nous verrons plus loin comment une certaine forme d'héritage multiple est néanmoins possible. Une classe dérivée hérite de tous les membres non privés de sa classe de base, mais elle peut redéfinir les méthodes dont elle hérite, ce qui permet le polymorphisme. En outre dans une méthode d'une classe dérivée, une nouvelle pseudo-variable, `super`, est définie. Elle permet:

– de forcer l'appel à une méthode définie dans la classe de base, quand celle-ci a été redéfinie dans la classe courante. Ceci est en particulier utile quand une redéfinition veut s'appuyer sur la définition de base.

– dans un constructeur, d'appeler un constructeur de la classe de base par une technique analogue à celle vue pour `this`:

```
super(args);
```

En fait tout constructeur d'une classe dérivée ne contenant pas d'appel explicite à `this(...)` ou à `super(...)` est supposé commencer par "`super();`", ce qui déclenchera une erreur du compîlateur si la classe de base ne possède pas de constructeur sans argument, explicite ou implicite.

En fait, toutes les classes autres qu'`Object` qui ne comportent pas de clause `extends` explicite sont automatiquement considérées comme ayant une clause "`extends Object`". Toutes les classes autres qu'`Object` sont donc des classes dérivées. Une classe peut par contre "résister" à sa propre dérivation:

– Déclarer la classe elle-même comme `final` interdit toute dérivation
– Déclarer une méthode non privée comme `final` interdit la redéfinition de cette méthode particulière

Enfin l'héritage permet des conversions, implicites ou explicites, entre les types références. De façon générale si `Base` est une classe, et `Derived` une classe dérivée de `Base`:

1. Une expression de type `Derived` peut être silencieusement convertie dans le type `Base`, ce qui permet à une variable déclarée `Base` de référence un objet de typ réel `Derived`, et à une méthode acceptant un argument déclaré `Base` de recevoir un argument de type réel `Derived`, ce qui est la base du polymorphisme.
2. Une expression de type `Base` peut être explicitement convertie dans le type `Derived` par la syntaxe:

   ```
   (Derived)expr
   ```

   qui sera acceptée par le compilateur, mais qui pourra donner lieu à une erreur à l'exécution, si la "promesse" correspondante n'est pas tenue. Ceci s'applique aussi au cas où `Base` est un nterface, et `Derived` est une classe ou un interface qui n'est *pas* explicitement dérivé de `Base`.
3. Une expression de type `Derived[]` peut être silencieusement convertie dans le type `Base[]`, mais c'est une conversion "dangereuse", qui peut amener indirectement une erreur ultérieure à l'exécution: en effet le code peut ensuite affecter à un élément du tableau converti une référence à un objet qui n'est pas une instance de `Derived`. Ceci est démontré dans

l'exemple (par ailleurs inintéressant!) suivant:

```
public class Bug {
  private static void spoil(Object[] arg) {
    if (arg.length>0)
      arg[0]=new Object();
  }
  public static void main(String[] arg) {
    spoil(arg);
  }
}
```

Ce code compile sans erreur ni avertissement, et l'exécuter par " `java Bug`" ne provoque aucun incident, mais exécuter:

```
java Bug quoi
```

provoque le message d'erreur suivant:

```
java.lang.ArrayStoreException:
  at Bug.spoil(Bug.java: 4)
  at Bug.main(Bug.java: 7)
```

En `C++`, ceci correspondrait au fait que si `Derived` dérive de `Base`, `Derived *` est acceptable comme `Base *`, mais que `Derived **` n'est *pas* acceptable comme `Base **`.

5.7 Accès

Il faut bien distinguer la visibilité de l'accessibilité, même si les diagnostics du compilateur ne le font pas toujours. Si un objet `obj` d'une classe `Clz` posséde un membre `elt`, hérité ou nom, ce membre est visible:

- comme `obj.elt` dans tous les cas
- comme `Clz.elt` si c'est un membre de classe, et c'est alors la notation conseillée.

Par contre, comme nous l'avons déjà décrit, le niveau de protection d' `elt`, et la relation entre la classe qui contient le code référençant et `Clz` peuvent en interdire l'usage. Dans le code d'une méthode de `Clz`, les notations `this.elt` ou `Clz.elt` peuvent être remplacées par la forme abrégée `elt`, dans la mesure où il n'y a pas de conflit avec le nom d'un argument ou d'une variable locale. Pour éviter les ambigüités sans alourdir le code, il est conseillé de donner aux attributs, surtout ceux d'instance, un nom reconaissable, comme, dans ce cas, `elt_` ou `m_elt`.

5.8 Abstraction

Bien que les niveaux de protection procurent une certaine séparation de l'interface et de l'implémentation, elle n'est pas totale, en particulier le source d'une classe mélange les deux. Il existe un équivalent "pur interface" d'une classe, introduit par le mot-clef `interface`, justement. La syntaxe de la définition d'un interface est voisine de celle d'une classe, avec quelques importantes différences:

- Un interface peut être public ou package, mais aucun qualificateur de protection ne peut être appliqué à ses membres qui sont tous implicitement publics
- Seules les méthodes peuvent être des membres d'instances, tout membre attribut est implicitement déclaré `final`
- Les méthodes n'ont pas de définition, la partie " `{...}`" est remplacée par "`;`". Elles sont "abstraites".
- Une classe qui veut "hériter" d'un interface déclare " `implements` *`interface`*" (en effet elle n'hérite que des déclarations, pas des définitions qu'elle devra fournir), et cette clause peut être multiple, c'est-à-dire qu'une classe peut implémenter plusieurs interfaces!
- Un interface ne peut bien sûr pas hériter d'une classe, pas même d' `Object`, mais il peut hériter d'un ou de plusieurs interfaces, en utilisant cette fois le mot-clef `extends`.

Cette notion d"'abstrait" est également étendue aux classes: une classe peut être déclarée `abstract`, ce qui indique qu'elle n'implémente pas toutes les méthodes qu'elle déclare (ou dont elle hérite). Les méthodes déclarées sans être implémentées doivent être elles-mêmes précédées du mot-clef `abstract`. Un cas typique de classe abstraite est une classe qui implémente partiellement un interface. Une classe abstraite est donc intermédiaire entre un interface et une classe "normale", mais une autre classe ne peut (à nouveau) dériver que d'une seule classe abstraite, ce qui limite leur emploi dans une certaine mesure.
On peut déclarer une variable avec pour type une référence à un interface ou à une classe abstraite, mais on est alors certain qu'à l'exécution, si cette variable contient autre chose que la référence `null`, elle dénote un objet ayant un autre type que le type déclaré!

6 La gestion des exceptions

6.1 Mécanismes généraux

De nombreuses erreurs peuvent survenir pendant l'exécution d'un programme, qui rendent impossible de pousrsuivre l'exécution comme si rien n'était arrivé. `java` définit pour cela la notion d'exception. Une exception provoque la terminaison abrupte du bloc de code en cours d'exécution, et le transfert du contrôle de l'exécution en un point spécifié par le programmeur comme traitant cette

exception. Les exceptions sont en fait des classe `java`, déclenchées soit implicitement par une erreur du programme, soit explicitement par l'emploi de la syntaxe:

```
throw new ExceptionClass(...);
```

dans du code ayant déterminé qu'il ne peut pas s'exécuter correctement dans le cas présent. Nous avons vu que le programmeur peut spécifier le traitement d'une exception par la syntaxe:

```
try tryBlock
catch(ExceptionClass id) handlerBlock
    ...
```

Le contrôle de l'exécution sera tranféré au début du bloc *handlerBlock* si une exception dérivée de la classe *ExceptionClass* est levée pendant l'exécution de *tryBlock*.

6.2 Particularités à la compilation

Les exceptions ne peuvent pas être n'importe quelles classes, elles doivent être dérivées de la classe `java.lang.Throwable`. La hiérarchie dérivée de cette classe est divisée en 3 parties:

1. La classe `java.lang.Error` et ses dérivées
2. La classe `java.lang.RunTimeException` et ses dérivées
3. Les autres classes dérivées de `java.lang.Exception`

Une méthode peut déclarer les méthodes que son exécution peut lever sans qu'elle les gère elle-même par la clause:

```
throws ExceptionClass1, ExceptionClass2
```

Une méthode *doit* déclarer de telles exceptions si elles sont du type 3. Cette obligation n'existe pas pour les autres types:

- Les dérivées de `Error` sont des erreurs difficilement récupérables et qui peuvent se produire n'importe où
- Les dérivées de `RunTimeException` sont des erreurs qui peuvent être levées par de nombreuses constructions du langage, mais dont le développeur peut se garantir par des techniques que le compilateur n'est pas en mesure de reconnaître: par ex. `ArithmeticException` (division par 0), `NullPointerException, ClassCastException` (toutes ces exceptions sont dans le package `java.lang`).

Exiger que tout programme déclare ou traite ces erreurs nuirait en définitive à la lisibilité et à la maintenabilité du code.

7 Les Composants `java`

7.1 Introduction

Au début, les seuls composants `java` étaient les classes dérivées de `java.awt.Component`, c'est-à-dire des composants graphiques interactifs, destinés à être programmés explicitement par le développeur. Le notion maintenant recouvre des composants qui peuvent ne pas avoir de "facette" interactive, mais surtout qui peuvent être manipulés par des outils de conception, voire interagir avec d'autres composants, qui ne les "connaissaient" pas au moment de la compilation. Ces mécanismes ont besoin de moyens de communication standardisés mais aussi extensibles.

7.2 Sérialisation

La capacité de sauvegarder l'état d'un objet et de le restaurer ultérieurement est intéressante en soi. Elle l'est particulièrement dans le cas de composants que des outils visuels permettent de paramétrer "sur-mesure", car ce paramétrage est perdu si on ne peut le sauvegarder. `java` prévoit un mécanisme pour sauvegarder un objet sur un flux de sortie (comme un fichier). Ce mécanisme est "à plusieurs vitesses", c'est-à-dire qu(il est plus ou moins automatique ou sous contrôle du programme suivant le désit du développeur. Une classe qui implémente l'interface `Serializable` profite au maximum de l'automatisation. Une certaine mesure de contrôle peut être amenée par un nouveau qualificateur pour les attributs, `transient`. Un attribut déclaré `transient` ne sera ni écrit, ni relu au moment de la sérialisation. Il y a plusieurs possibles pour déclarer un attribut `transient`:

- C'est un attribut temporaire, calculé à partir des autres pour faciliter ou accélérer certaines opérations (par exemple un cache).
- C'est un attribut qui n'a de sens qu'à l'exécution (par exemple une connection base de données).
- C'est une référence sur un objet qui n'est pas sérialisable. Bien sûr, si connaître son état est indispensable pour reconstituer l'état de l'objet courant, celui-ci n'est pas non lus sérialisable!
- C'est une attribut "sensible", au sens de la sécurité. Après tout, écrire tous ses attributs sur un flux de sortie sur simple demande est un peu à contre courant de la sécurité (de l'encapsulation aussi, si l'on y réfléchit!).

Mais si la classe veut être sérialisable, mais en contrôlant totalement ce qui est écrit et relu, et dans quel format, elle doit implémenter l'interface `Externalizable`, et fournir les méthodes `writeObject` et `readObject`.

7.3 Gestion des événements

Les événements sont un des principaux moyens de communication des composants, tels qu'ils soient, entre eux. Les composants ayant de plus en plus d'importance dans `java`, le modèle événementiel est d'autant plus important. Très re-

manié également depuis jdk1.0, initialement défini dans `java.awt`, les interfaces de base sont maintenant dans `java.util` (pourquoi pas?), avec les implémentations prédéfinies interessantes partagées entre `java.beans` et `javax.swing`. Le modéle relève du principe abonnement/notification. Il y a 3 type d'acteurs, ayant des relations logiques très circulaires:

1. Les événements eux-même. Ce sont des classes dérivées de `java.util.EventObject`, c'est une classe concréte, mais sa seule propriété est sa source. Des objets de cette classe peuvent être instantiés et transmis, mais ils ne comportent aucun état, ne transportent aucune autre information que celle d'avoir été suscités, et par qui.
2. Les "listeners". Il existe un interface `java.util.EventListener`, mais il est vide. Pour chaque classe spécifique d'événement, on définit un interface qui étend `EventListener`, et qui contient une méthode pour traiter l'événement en question. Par exemple, un événement trés populaire est `java.awt.event.ActionEvent`. Cet événement correspond à l'action de l'utilisateur sur un bouton, ou un item de menu, et contient (entre autres) une variable d'état représentant la commande sous forme d'une chaîne de caractères. Il lui correspond, dans le même package, un listener:

   ```
   public interface ActionListener extends EventListener {
     void ActionPerformed(ActionEvent e);
   }
   ```

   Une implémentation concrète de ce listener pourra agir en fonction de la propriété `actionCommand` de l'`ActionEvent`.
3. Le source d'un événement. Il n'y a pas de classe ni d'interface de base, à part `Object!` Une source est un génère des événements, elle doit donc connaître certains types d'évènements , donc les types de listener correspondants et doit définir des méthodes :

   ```
   addXEventListener(xEventListener listener);
   ```

   et

   ```
   removeXEventListener(xEventListener listener);
   ```

   rien n'interdit à un objet de, ni ne le force à, être simultanément source et listener. Des sources typiques sont:

   - des composants graphiques "classiques", dans `java;awt` ou `javax.swing`. Par exemple les boutons sont des sources de `ActionEvent`.
   - des javabeans

7.4 Les Java Beans

La notion de JavaBean est assez informelle: il n'y a pas d'interface de ce nom à implémenter obligatoirement. Presque n'importe classe publique `java` peut être un `bean`, à condition de respecter un petit nombre de contraintes, au moins si elle veut être un `bean` "intéressant":

- Un `bean` qui se respecte doit être sérialisable, c'est à dire qu'il doit implémenter soit `Serializable`, soit `Externalizable` (tous les deux, dans le package `java.io`). Un `bean` est censé être instantié par des méthodes de classe particulères de `java.beans.Beans`, qui chercheront à rétablir, par désérialisation, un état antérieur conservé par sérialisation.
- Un `bean` doit posséder un constructeur public appelable sans argument. L'instantiation ci-dessus n'a pas d'argument particulier à passer à un constructeur.
- Un bean doit posséder un certain nombre de méthodes publiques "reconnaissables" par les outils grâce à un mécanisme appelé "introspection". Une métaclasse d'information peut être utilisée pour publier ces méthodes, mais par défaut des règles de nommage simple sont appliquées, ce qui suit se réfèrera à ce cas simplifié. Les méthodes reconnues par les outils sont de plusieurs types:
  - Les propriétés: un `bean` exporte une propriété `name` de type `type` s'il possède au moins une des deux méthodes publiques suivantes:
    - `type getName();`
    - `void setName(type value);`

    S'il possède ces deux méthodes, la propriété est accessible en lecture/écriture, si seule la méthode `get` existe, la propriété est accessible en lecture seule, si seule la méhode `set` existe, la propriété est accessible en écriture seule. Si `type` est `boolean`, la méthode `get` peut être remplacée par `isName()`
  - Les événements (voir plus haut la description des sources): Un bean annonce qu'il est une source d'événements `EventName` s'il possède deux méthodes publiques:
    - `addEventNameListener(EventNameListener listener)` , accompagnée ou nom d'une clause `throws TooManyListeners`
    - `removeEventNameListener(EventNameListener listener);`
  - Les deux peuvent se combiner: deux évènements liés aux propriétés sont prédéfinis: `PropertyChange` et `VetoableChange`. Le premier sert à définir la notion de "propriété liée", c'est-à-dire de propriété dont il est possible de demander à être notifié des changements. Le second sert à définir la notion de "propriété contrainte", c'est-à-dire de propriété aux changements de laquelle il est possible de s'opposer.

Un bean qui possède une propriété accessible en écriture dont il veut faire une propriété liée doit se signaler comme source d'événements `PropertyChange` (il existe des versions des fonctions `add/remove` avec un argument supplémentaire pour spécifier le nom de la propriété), et bien sûr, générer les éléments dans la méthode `set`. Une classe auxiliaire, `PropertyChangeSupport` existe pour faciliter cette implémentation (toutes les classes et interfaces mentionnés ici sont dans le package `java.beans`). Un mécanisme semblable est défini pour les propriétés contraintes.

  - Les méthodes publiques dont le nom n'est pas conforme à un des schémas que nous venons de voir sont simplement reconnues comme méthodes accessibles.

- En fait, les spécifications dec beans liées au jdk1.2 ont enrichi ce modèle et en particulier, spécifient un interface, `BeanChild`, qu'un bean est encouragé à implémenter pour profiter de toutes les nouveautés apportées par 1.2. Néanmoins, pour des raisons de compatibilité, il n'est pour l'instant pas envisagé de rendre l'implémentation de cet interface obligatoire.

Bien que cette liste soit longue, ce qu'un bean est *obligé* d'implémenter est limité, car il peut exporter n'importe quel nombre de propriétés, y compris 0, et de même pour les événements. En fait dans un cas limite d'utilisation des servlets de WebSphere (extension IBM pour serveur HTTP), avec une définition pré-jdk1.2 des Java Server Pages, il est possible d'utiliser comme bean... un `String`! Mais si le bean a été conçu pour interopérer avec d'autres, un outil de manipulation visuelle, comme la BeanBox de Sun, peut présenter ses propriétés à éditer, puis les sauver par sérialisation, peut assembler plusieurs beans, et connecter des événements générés par l'un à des méthodes publiques de l'autre, tout cela avec un pilotage purement visuel du développeur.

8 Programmation `java` avancée

8.1 Introduction

L'un des attraits que possédait `java` dès sa sortie, même en version jdk1.0beta, c'est qu'il était accompagné de librairies qui rendaient extrêmement facile de développer des applications qui aurient mérité le qualificatif d'avancé avec tout autre environnement (par exemple une maquette de serveur HTTP). En particulier, le support de la programmation réseau, et de la concurrence, étaient remarquable d'entrée par leur facilité d'utilisation.

8.2 Les threads et le multithreading

La concurrence est représentée en `java` par deux éléments:

- Une classe prédéfinie, `Thread`, qui modélise ce qu'on appelle aussi un "processus léger", accompagnée d'une classe auxiliaire `Runnable`.

– Un mot-clef particulier, `synchronized`, qui prend en charge tous les besoins de synchronisation

8.2.1 Qu'est ce qu'un thread

On appelle thread en général (pas spécialement en `java`) un processus exécutable interne à un processus système. Un nombre potentiellement illimité de threads peuvent coexister au sein d'un processus systèmes. Ils ont de gros avantages en ce qui concerne les performances par rapport la multiplication des processus "lourds" que sont les processus systèmes:

1. Ils partagent à priori leurs ressources, en particulier l'accès à la mémoire (ce qui présente d'autres inconvénients)
2. Ils peuvent s'allouer des ressources privées si cela est nécessaire
3. La transition entre deux threads d'un même processus est peu coûteuse

La contrepartie de 1 est que le système ne fournit aucune protection directe contre des utilisations asociales de ses ressources. Une programmation multithread demande donc en général une gestion attentive de la synchronisation des accès à ces ressources partagées.

8.2.2 Les `Thread java`

Créer une instance de la classe `Thread` est la seule façon de lancer l'exécution d'une fonction dans un nouveauprocessus léger. Il y a deux méthodes, légèrement différentes:

1. Ecrire une classe qui dérive de `Thread`, et redéfinir la méthode `run()`:

   ```
   public class Task extends Thread {
     public void run() {
       // code à exécuter
     }
   }
   ```

   puis lancer l'exécution par

   ```
   new Task().start();
   ```

2. Ecrire une classe qui implémente l'interface `Runnable`, et définir la méthode `run()`:

   ```
   public class task implements Runnable {
     public void run() {
       // code à exécuter
     }
   }
   ```

puis lancer l'exécution par:

```
new Thread(new Task()).start();
```

Bien sûr, une référence sur le `Thread` peut être conservée, mais ce n'est pas nécessaire, un `Thread` actif est référencé par le système, et ne sera pas "Garbage collecté" tant que sa méthode `run()` s'exécute.

8.2.3 Les difficultés liées à la concurrence

La difficulté de la conception d'une application multithread réside dans le fait de devoir prendre en compte les problèmes de synchronisation des threads. D'une façon générale, ces problèmes rentrent dans deux catégories:

1. L'exclusion mutuelle, qui correspond à la notion de sections critiques, c'est-à-dire de traitements dont on veut garantir qu'un seul `Thread` peut les exécuter à la fois. Il s'agit par exemple de modifications de données partagées entre les threads
2. La notion d'événement/notification, qui correspond qu'un peut se mettre en attente d'un signal de la part d'un autre. Il s'agit par exemple du cas d'un `Thread` consommant des données produites par un autre `Thread`.

En `java`, ce seront des objets qui joueront aussi bien le rôle de verrou d'exclusion mutuelle (moniteur) que d'événement. En fait un objet ne peut jouer le rôle d'événement que s'il joue déjà celui de moniteur.

8.2.4 L'exclusion mutuelle

Pour mettre en évidence ce problème, considérons simplement la très utilisée classe `java.util.Vector`. Supposons qu'une variable de type `Vector`, disons `elements`, soit visible par deux sections de code exécutés par deux threads différents:

1. code exécuté par le 1er `Thread`:

```
Object elt=null;
if (elements.size()>0)
  elt=elements.elementAt(0);
```

2. code exécuté par le 2ème `Thread`:

```
if (elements.size()>0)
  elements.removeAt(0);
```

En l'absence de toute synchronization, l'ordonnancement des instructions entre les deux threads est libre. Dans le cas où les deux threads arrivent au début des sections de code alors que `elements` contient exactement *un* élément, les deux appels à `size()` peuvent renvoyer 1, puis l'appel à `removeElementAt` peut

s'exécuter en premier, suivi par l'appel à `elementAt` qui déclenchera maintenant une exception, malgré la précaution prise de tester `size()` auparavant. Le mot-clef `synchronized` sert à résoudre ce problème. On réécrit les deux sections:

1. code exécuté par le 1er `Thread`:

```
Object elt=null;
synchronized(elements) {
if (elements.size()>0)
  elt=elements.elementAt(0);}
```

2. code exécuté par le 2ème `Thread`:

```
synchronized(elements) {
if (elements.size()>0)
  elements.removeAt(0);}
```

L'effet de `synchronized` est d'acquérir le verrou lié à l'objet `elements` pour la durée du bloc entre accolades. Maintenant les deux threads ne pourront pas pénétrer en même temps dans les sections synchronisés, le deuxième arrivé devra attendre que le premier en soit sorti.

Ceci est l'usage le plus général de synchronized, mais l'usage le plus fréquent est en fait un cas particulier, où `synchronized` est utilisé comme qualificateur d'une méthode. Si la méthode est une méthode d'instance, ceci est équivalent à encadrer le code par la méthode par `synchronised(this){...}`. Deux méthodes d'instance synchronisées ne peuvent pas s'exécuter en parallèle sur un même objet, c'est-à-dire que l'exécution de l'une précèdera totalement l'exécution de l'autre. Si `synchronized` est appliqué à une méthode de classe, l'effet est de synchroniser sur l'unique objet représentant la classe. Deux méthodes de (la même) classe synchronisées ne peuvent pas s'exécuter en parallèle.

Un `Thread` peut acquérir plusieus fois le même verrou. Une méthode `synchronized` peut donc en appeler une autre, sans provoquer de deadlock, le verrou ne sera relâché qu'après la terminaison du bloc `synchronized` le plus externe. Bien sûr, les blocs `synchronized` qui sont terminés par une exception sont gérés proprement.

Pour des raisons analogues à celles rencontrées dans l'exemple précédent, la plupart des méthodes d'instance de `Vector` sont déjà synchronisées. Néanmoins, la synchronisation n'est pas gratuite en termes de performance. C'est pourquoi les nouvelles classes de `Collection` du `jdk1.2` ne sont plus synchronisées par défaut.

D'autre part l'accès a des attributs individuels d'un objet peut être garanti des problèmes de synchronisation à moindre coût, grâce aux deux points suivante:

– L'accès à une variable individuelle est atomique, c'est-à-dire que, par exemple, un `Thread` qui lit une variable de type `short` dans une méthode

non synchronisée ne risque pas de recevoir un monstre dont 8 bits contiendraient ce qu'il y avait mis à son dernier accès, et les 8 autres ce qu'un autre `Thread` vient d'y mettre. Attention, ceci n'est pas garanti pour les variables sur 64 bits (`long` et `double`)!

– Si un attribut est déclaré `volatile`, chaque lecture est garantie renvoyer son état le plus récent, c'est-à-dire que toute modification effectuée par un `Thread` est immédiatement visible par les autres. Sans ce qualificateur, la machine virtuelle n'est absolulment pas tenue de garantir ce comportement.

8.2.5 Attente et notification d'événement.

Si un `Thread` a acquis le verrou associé à un objet, la méthode `wait` peut être invoquée sur cet objet dans le contexte de ce `Thread`. Celui-ci est alors mis en attente (après avoir temporairement libéré le verrou), jusqu'à ce qu'un `Thread` actif, et ayant à son tour acquis le verrou, invoque sur cet objet la méthode `notify`, `notifyAll`, ou après un time-out si l'invocation de `wait` en a spécifié un. La méthode `notify` sélectionne au hasard un `Thread` parmi ceux en attente sur l'objet pour le réactiver, alors que dans les autres cas tous les threads concernés le sont. Même si le `Thread` est choisi, il doit pour devenir effectivement actif obtenir de niveau le verrou, et d'autres threads peuvent être en compétition (en particulier le `Thread` ayant invoqué `notify` qui avait acquis le verrou à ce moment, s'il ne l'a pas relâché depuis).

1. Voici un exemple simple, mettant en jeu un producteur et un consommateur. La classe `SharedData` définit une donnée partagée, ici un simple entier. Le consommateur peut lire la donnée par la méthode `getValue`, le producteur peut lk'écrire par la méthode `putValue`. La synchronisation est double: le consommateur ne peut récupérer la donnée (une fois) que si le producteur vient de la mettre à disposition, et le producteur ne peut

déposer une donnée que si la précédente a été consommée.

```
public class SharedData {
  private int contents_;
  private boolean available_=false;
  public synchronized int getValue() {
    while (!available_) {
      try {
        wait();
      } catch(InterruptedException e) {
      }
    }
    available_=false;
    notifyAll();
    return contents_;
  }
  public synchronized void setValue(int value) {
    while (available_) {
      try {
        wait();
      } catch(InterruptedException e) {
      }
    }
    available_=true;
    contents_=value;
    notifyAll();
  }
}
```

2. Voici maintenant un exemple plus complexe, permettant à un `Thread` d'acquérir une ressource en exclusivité pour la modifier, mais permettant un accès partagé en lecture aux autres threads. La classe `Lock` ci-dessous

implémente un verrou de ce type.

```
public class Lock {
  private int shared_ = 0;
  private boolean exclusive_ = false;
  public synchronized void acquireShared() {
    if (exclusive_) {
      while (exclusive_)
        try {
          wait();
        } catch(InterruptedException e) {
        }
    }
    shared_++;
  }

  public synchronized void release() {
    if (shared_>0) {
      shared_--;
      if (shared_==0)

notify();  // Notify to one waiting exclusive Thread
    }
    else if (exclusive_) {
      exclusive_ = false;

notifyAll(); // Notify to all waiting shared threads
    }
  }
  public synchronized void acquireExclusive() {
    if (exclusive_ || shared_>0) {
      while (exclusive_ || shared_>0)
        try {
          wait();
        } catch(InterruptedException e) {
        }
    }
    exclusive_ = true;
  }
}
```

et voici la donnée partagée:

```
public class SharedData {
  private int contents_;
  private Lock theLock_=new Lock();
  public int getValue() {
    int value=0;
    try {
      theLock.acquireShared();
      value=contents_;
    } finally {
      theLock.release();
    }
    return value;
  }
  public void setValue(int value) {
    try {
      theLock.acquireExclusive();
      contents_.setValue(value);
    } finally {
      theLock.release();
    }
  }
}
```

Cette fois, la donnée n'est pas "consommée" par la lecture.

8.3 La programmation réseau

8.3.1 Introduction

Les classes du package `java.net` contiennent principalement:

- Des classes URL permettant d'accéder au contenu d'une ressource WEB
- Des classes correspondant à la notion de socket pour la communication programme à programme.

Les classes socket permettent une communication programme à programme aussi bien en mode datagramme qu'en mode connecté. Nous reprendrons dans ces deux cas un même exemple de couple client serveur, le serveur se bornant ici à faire l'écho de chaînes de caractères envoyées par le ou les clients.

8.3.2 `Socket` en mode connecté

En mode connecté, on commence d'abord par créer un canal de communication entre les deux applications. Le mécanisme est dissymétrique, le client utilise

un objet `Socket` pour établir un connexion avec un serveur qui utilise de son côté un ServerSocket pour recevoir ces demandes de connexion. Un fois la connexion établie, on dispose d'un `InputStream` et d'un `OutputStream`, respectivement pour lire ou écrire dans le socket. Dans notre exemple, de façon à pouvoir servir plusieurs clients simultanément, le serveur crée un `Thread` pour chaque nouveau client. Le serveur crée un objet de type `ServerSocket`, le constructeur a comme argument la porte sur laquelle le socket est à l'écoute. La méthode accept est bloquante jusqu'à la connexion d'un nouveau client, elle retourne un nouveau socket ouvert avec le client, ce socket est transmis au constructeur du `Thread` de service qui crée un `BufferedReader` et un `PrintWriter`, respectivement pour lire et écrire dans le socket.

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

```
class ServiceThread extends Thread {
  private final Socket socket_;
  private final BufferedReader in_;
  private final PrintWriter out_;
  ServiceThread(Socket socket) throws IOException {
    socket_ = socket;
    in_ = new BufferedReader(
           new InputStreamReader(
             socket_.getInputStream()));
    out_ = new PrintWriter(
            new BufferedWriter(
              new OutputStreamWriter(
                socket_.getOutputStream())), true);
  }
  public void run() {
    try {
      while(true) {
        String str = in_.readLine();
        if (str == null)
          break;
        out_.println(str.toUpperCase());
      }
    }
    catch(IOException e) {
    }
    finally {
      System.out.println("Connection closed");
      try {
        socket_.close();
      }catch(IOException e) {
      }
    }
  }
}

public class ThreadedEchoServer {
```

```
public static void main(String[] args) throws IOException {
    int port=0;
    try {
      port = Integer.parseInt(args[0]);
    } catch (Exception e) {
      System.out.println("Usage: EchoServer port");
      System.exit(1);
    }
    try {
      ServerSocket listen = new ServerSocket(port);

System.out.println("Listening on socket " + listen);
      try {
        while(true) {
          Socket socket = listen.accept();

System.out.println("New connection " + socket);
          try {
            new ServiceThread(socket).start();
          } catch(IOException e) {
            socket.close();
          }
        }
      } finally {
        listen.close();
      }
    } catch(IOException e) {
      System.out.println(e);
    }
  }
}
```

Du coté client, le constructeur du socket comporte comme argument l'adresse Internet de la machine serveur ainsi que la porte sur laquelle écoute le serveur. L'adresse Internet est obtenue à partir du nom de la machine serveur par la méthode `getByName` qui fait appel au service de nom. De la même façon que le serveur, le client crée des stream pour lire et écrire dans le socket.

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

```
public class EchoClient {
  public static void main(String[] args) {
    int port=0;
    String host=null;
    try {
      port = Integer.parseInt(args[1]);
      host = args[0];
    }
    catch (Exception e) {
      System.out.println("Usage: EchoClient host port");
      System.exit(1);
    }
    try {
      InetAddress address = InetAddress.getByName(host);
      Socket socket = new Socket(address, port);
      try {
        System.out.println("Socket " + socket);
        BufferedReader in = new BufferedReader(
          new InputStreamReader(
            socket.getInputStream()));
        PrintWriter out = new PrintWriter(
          new BufferedWriter(
            new OutputStreamWriter(
             socket.getOutputStream())), true);
        BufferedReader sysin = new BufferedReader(
          new InputStreamReader(System.in));
        while(true) {
          String str = sysin.readLine();
          if(str == null)
            break;
          out.println(str);
          str = in.readLine();
          System.out.println(str);
        }
      }
      finally {
        socket.close();
      }
    }
    catch(IOException e) {
      System.out.println(e);
    }
  }
}
```

8.3.3 Socket en mode datagramme

Un socket de type `DatagramSocket` permet l'envoi ou la réception de paquets de données. Coté émission, on construit un paquet de données `DatagramPacket` à partir du contenu du message, de l'adresse de destination (un objet de type `InetAdress`) et de la porte de destination. On peut ensuite transmettre le message par la méthode send sur un `DatagramSocket`. Coté réception, on construit également un `DatagramPacket` que l'on fournit comme argument à la méthode `receive` sur un `DatagramSocket`.

Le serveur d'écho en mode datagramme:

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

public class DatagramEchoServer {
  public static void main(String[] args) {
    int port=0;
    try {
      port = Integer.parseInt(args[0]);
    }
    catch (Exception e) {
      System.out.println("Usage: EchoServer port");
      System.exit(1);
    }
    try {
      DatagramSocket socket = new DatagramSocket(port);
      byte[] buffer = new byte[1000];
      DatagramPacket inpacket= new DatagramPacket(
        buffer, buffer.length);
      while(true) {
        socket.receive(inpacket);

System.out.println("Message from " +                    inpacket.getAddress().getHostl

inpacket.getAddress() + ") port " + inpacket.getPort());

String str = new String(buffer,0,inpacket.getLength());
        str = str.toUpperCase();
        DatagramPacket outpacket = new DatagramPacket(
          str.getBytes(), str.length(),
```

```
inpacket.getAddress(), inpacket.getPort());
        socket.send(outpacket);
        inpacket.setLength(buffer.length);
      }
    }
    catch(IOException e) {
      System.out.println(e);
    }
  }
}
```

Le client en mode datagramme:

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

```
public class DatagramEchoClient {
  public static void main(String[] args) {
    int port=0;
    String host=null;
    try {
      port = Integer.parseInt(args[1]);
      host = args[0];
    }
    catch (Exception e) {
      System.out.println("Usage: EchoClient host port");
      System.exit(1);
    }
    try {
      InetAddress address = InetAddress.getByName(host);
      DatagramSocket socket = new DatagramSocket();
      byte[] buffer = new byte[1000];
      DatagramPacket inpacket = new DatagramPacket(
        buffer, buffer.length);
      BufferedReader sysin = new BufferedReader(
        new InputStreamReader(System.in));
      while(true) {
        String str = sysin.readLine();
        if(str == null)
          break;
        DatagramPacket outpacket = new DatagramPacket(
          str.getBytes(), str.length(),
          address, port);
        socket.send(outpacket);
        socket.receive(inpacket);
        str = new String(buffer,0,inpacket.getlength());
        System.out.println(str);
        inpacket.setlength(buffer.length);
      }
      socket.close();
    }
    catch(IOException e) {
      System.out.println(e);
    }
  }
}
```

9 La sécurité

9.1 Les mécanismes de la sécurité

Plusieurs mécanismes interviennent dans le maintien de la sécurité:

1. Au niveau le plus bas, le compilateur et le langage garantissent que le code exécuté ne peut pas faire exploser la machine virtuelle, mais peut tout au plus lever des exceptions "propres".
2. Nous avons vu tout à l'heure les mécanismes de la vérification de pseudo-code.Le vérificateur protège la machine virtuelle contre du pseudo-code invalide, peut-être malicieux, qui pourrait déclencher l'exécution de zônes de code réservées.
3. Les chargeurs de classe sont un autre mécanisme sécurisant: Un chargeur de classe est un objet `java` responsable de "trouver" une classe étant donné une référence symbolique à résoudre, et peut-être de la charger. Une table est maintenue par la machine virtuelle, dont les clefs sont des couples d'un nom complet de classe et d'une référence sur un chargeur de classe, et dont les data sont des classes `java` chargées (les classes `java` représentées en mémoires par des objets `java` de classe `java.lang.Class`). Toute classe contient une référence au chargeur de classe qui l'a chargé, et donc qui l'a instantié. Quand une référence symbolique sur une classe est rencontrée dans du code appartenant à une classe déjà présente, la procédure suivante est observée pour la résoudre:

   (a) On cherche dans la table ci-dessus s'il existe une entrée dont la clef est la combinaison du nom constituant la référence symbolique et du chargeur de la classe courante. Si c'est le cas, la classe correspondante est utilisée *sans appeler aucune méthode du chargeur de classe*. Ceci a deux conséquences:

      i. Le chargeur de classe ne peut pas pervertir le système en résolvant différemment une même référence symbolique en deux occasions.
      ii. Si une classe ayant le même nom est déjà présente mais associée à un autre chargeur, elle ne sera pas forcément utilisée dans ce contexte. Ceci évite qu'une classe chargée par exemple par un `applet` malveillant et possédant le même nom qu'une classe système non encore chargée soit trouvée et appelée par du code "sain".

   (b) Si la classe n'a pas été trouvée en (a), alors la méthode `loadClass` est invoquée sur le chargeur de la classe courante. Quand cette méthode retourne avec succès, le résultat est utilisé pour créer une nouvelle entrée dans la table, avec le nom de la classe, et ce chargeur. Ceci n'implique pas néanmoins que ce chargeur soit celui associé;à la classe, car rien un chargeur peut demander à un autre chargeur de coopérer avec lui pour trouver une classe, si la politique de sécurité ne s'y oppose pas.

Au lancement de la machine virtuelle, il existe un seul chargeur de classes, appelé le chargeur primordial (qui n'est pas réellement un objet `java` mais plutôt partie intégrante de la machine virtuelle), qui est utilisé pour charger les classes locales, et en particulier les classes des packages de base. Plus tard d'autres chargeurs de classe pourront être instantiés applicativement, en particulier pour aller chercher des classes sur le réseau. Un chargeur différent est créé pour chaque serveur source de classes. La methode `loadClass` de ces chargeurs commence normalement par demander la coopération du chargeur de classe primordial. C'est ce qui fait que des applets en provenance d'URLs différents ne partageront pas des classes applets même si elles ont le même nom, mais que tous utiliserons la même classe `String`.

4. La politique de sécurité. C'est un objet global qui réalise un "mapping" entre un `CodeSource` et un ensemble de permissions. Un `CodeSource` représente l'origine d'une classe, en un sens similaire, mais plus riche que celui utilisé par les chargeurs de classes, puisqu'il combine URL originateur et (éventuel) certificat de sécurité. Les permissions couvrent tous les domaines sensibles de l'activité d'une application, comme l'accès aux fichiers, l'accès au réseau, l'ouverture de fenêtres, etc... La politique de sécurité est un nouveau mécanisme du jdk1.2, qui permet un contrôle plus fin de la sécurité, sans avoir besoin de le programmer, car une politique peut être spécifiée par un fichier texte, lui-même aisément configurable par un outil spécialisé, `policytool`.
5. Le gestionnaire de sécurité. C'est encore un objet global dont les méthodes vérifient que l'application n'outrepasse pas les permissions attribuées par la politique de sécurité en fonction des `CodeSource`. Quand on lance une machine virtuelle depuis la ligne de commande, il n'y a pas de gestionnaire de sécurité global tant que le code applicatif n'en installe pas un, ce qui fait que la politique de sécurité n'est pas appliquée! Bien sûr, un browser en installe un. Jusqu'au jdk1.1, c'était une classe abstraite qu'il était nécessaire d'implémenter, mais avec l'introduction de la politique de sécurité, la classe de base est parfaitement utilisable telle quelle.
6. Nous avons mentionné les certificats de sécurité en 5. Ils annoncent les mécanismes "extérieurs" de la sécurité, avec les archives et les cabinets, pour empaqueter ensemble classes et certificats, la cryptographie, pour garantir la valeur du résultat, les couches sécurisées de transport, etc...

9.2 L'évolution du modèle

1. Dans les premières versions, le modèle de sécurité était géré en "tout-ou-rien", par exemple, dans le jdk1.0:

   - Le code local, c'est-à-dire chargé depuis le disque par le chargeur de classes primordial, est considéré sûr, et n'est soumis à aucune restriction

- Le code lointain, c'est-à-dire tout le reste, par exemple des applets chargés sur le web, était suspect, et n'était autorisé à accéder à aucune resource extérieure: système de fichiers, exécution d'autres applications, etc... C'est ce qu'on appelle le sandbox.

2. Avec le jdk1.1, le modèle est toujours en tout ou rien, mais la frontière est plus souple: les certificats sont introduits, si un applet est "signé" et accompagné d'un certificat, celui-ci est montré à l'utilisateur, qui peut décider de s'y fier ou non. Suivant sa décision l' applet tournera en dehors ou en dedans du sandbox.
3. Avec le jdk1.2, le modèle est maintenant totalement déclinable. Les classes systèmes ont toujours toutes les permissions, mais en dehors du cas où aucun gestionnaire de sécurité n'est installé, le code de toute autre origine peut recevoir des droits opération par opération. La toute puissance des classes systèmes ne doit pas être interprétée hâtivement: le code qui s'exécute à un moment donné n'a une permission que s*i tous les intervenants dans la pile des appels possédent cette permission*. Cette remarque doit à son tour être nuancée, dans deux sens opposés:

   (a) Si du code est exécuté pour le compte d'un autre `Thread`, cette exigence sera étendue à la pile des appels du `Thread` client
   (b) Du code autorisé peut explicitement demander à rélaxer cette exigence vis-à-vis de ses appelants.