Java - Syntaxe de base

Un Abrégé

Eric Lefrançois

25 mars 2002
Version 1.0

Hes·so Haute Ecole Spécialisée de Suisse Occidentale
eiu
tcom

Contenu

En préliminaire

Tous les points qui s'appliquent à C ++ comportent la mention C ++ .

Les points qui s'appliquent à Java comportent la mention Java .

On trouvera dans cet abrégé l'essentiel de la syntaxe de base du langage Java. Ne sont pas traités :

- Héritage et polymorphisme,
- Objets actifs (concurrence),
- Concept d'interface,
- Applets,
- Librairies du JDK

---

1 Conventions d'écriture

Java

- On utilise plutôt des minuscules

- Les noms commencent par des minuscules, sauf :
  identificateurs de classe : `Frame`, `Button`, `String`, ..
  constantes symboliques : `static final double PI = 3.14 ;`

- Noms composés : getBack, taxMan, comeTogether, ..

- Les *chaînes de caractères* doivent être écrites sur une seule ligne

```
System.out.println ("Cou
                                Cou");    // Refusé !!
```

- Concaténation de textes: symbole « **+** »

```
System.out.println ("Cou" +
                           "Cou");// OK
int age = 20;
System.out.println ("J'ai " + 20 + "ans");
```

- Les caractères sont codés sur 16 bits !
  Il s'agit du jeux de caractères UNICODE qui comprend la plupart des alphabets du monde. Ainsi, il est possible d'écrire les identificateurs avec des lettres accentuées : `ouvrirFenêtre`, `commencerLaFête`, ..

Ne pas écrire le nom des classes avec des accents. Chaque classe correspondra à un fichier (un fois le programme compilé), et les systèmes d'exploitation actuels ne reconnaissent pas forcément les accents dans les noms de fichiers.

- Commentaires

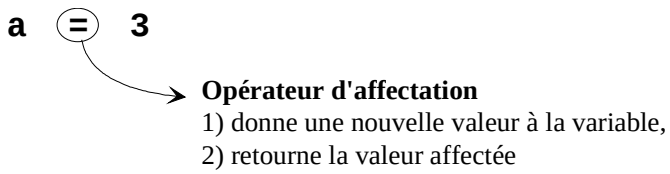
```
// commentaire court se terminant à la fin de la ligne

/*
Commentaire long.
Imbrication non possible.
A n'utiliser que pour masquer des parties de programme
(debugging), ou pour l'entête.
*/
```

Un commentaire long peut être utilisé aussi pour des commentaires très très courts au milieu d'une instruction !

```
⇨ System /* blabla*/ .out.println ("bla bla");
```

2 Instruction d'affectation

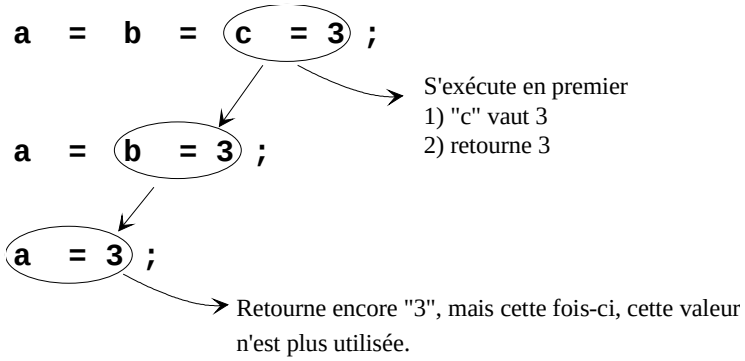
C ++
Java



Exemple:

```
ada → a := b := c := 3 ;     → INCORRECT
java →a = b = c =3 ;         → CORRECT
```

En Java, l'opérateur d'affectation a une « associativité » de type « R » (voir [Annexe A]) : cet opérateur s'exécute donc de la droite (Right) vers la gauche.



3 Opérateurs «++» et «--»

C ++
Java

++a a++	Augmente « a » de « 1 »
--a a--	Diminue « a » de « 1 »

A l'évaluation de ces opérateurs, deux actions sont réalisées :

1/ ils modifient la valeur de la variable,
2/ ils retournent une valeur

++a --a	retournent la **nouvelle** valeur de «a»
a++ a--	retournent **l'ancienne** valeur de «a» (avant l'opération)

Exemples :

```
a = 1 ;
b = a++ ;
c = ++a ;
c = a++ + ++a + b + c;
Que vaut a ? ⇨    5
Que vaut c ? ⇨    12
```

Pour déterminer les nouvelles valeurs de «a» et de «c», il nous faut connaître les règles de priorité des opérateurs (voir [Annexe A]):

Opérateur	Priorité Java *(le « 1 » est le plus prioritaire)*
++	1R
/	3L
+	4L

4 Opérateurs d'affectation composée «+=» «-=» «*=» ...

C ++
Java

Exemples

```
a *= 2 ⇔ a = a * 2
a += 4 ⇔ a = a + 4
```

Forme générale

```
a op= b  ⇔ a = a op b
```

où «**op**» peut valoir «+» «-» «*» «/» «%» etc..

Ces opérateurs retournent la nouvelle valeur de la variable affectée.

5 Opérateurs d'égalité «==»

C ++
Java

Exemple : `if (x==0) …..`

Cet opérateur retourne un bouléen : «`true`» ou «`false`» .

Notons qu'à la différence de Java, le type booléen n'existe pas en C++

En C++, l'opérateur d'égalité retourne `0` (⇔ « `false` ») ou `1` (⇔ « `true` »).

6 Autres opérateurs logiques

C ++
Java

Tous les opérateurs qui suivant retournent le bouléen «`true`» ou «`false` »:

		Exemple	*Equivalent Ada*
`!=`	différent	`if (x != 0) ..`	`/=`
`&&`	ET logique	`if (a>b && b>c) …`	`and then`

`||`	OU logique	`if (a<10 || a>20)..`	`or else`
`&`	ET	`if (a>b & b>c) …`	`and`

`>`
`<`
`>=`
`<=`
	comparaison	`if (x >= 3) …`	`>`
			`<`
			`>=`
			`<=`

Java

`|`	OU	`if (a<10 | a>20)..`	`or`
`!`	négation	`if ( !(x==0)) ..`	`not`

7 Opérateurs arithmétiques

C ++
Java

		Exemple	*Résultat*	*Equivalent Ada*
`+`	addition	`3 + 5`	`8`	`+`
`-`	Soustraction	`5 - 3`	`2`	`-`
`*`	Multiplication	`5 * 3`	`15`	`*`
`/`	Division	`5 / 3` `5.0 / 3.0`	`1` `1.66666…`	`/`
`%`	Modulo (reste de la division entière)	`5 % 3`	`2`	`mod, rem`

Java

`+`	concaténation	`"Hello" + "Martin"` `123 + ""`	`"Hello Martin"` `"123"`	`&`

L'opérateur « + » est utilisé pour concaténer les chaînes de caractères. Additionner un nombre avec une chaîne vide est une astuce qui permet très facilement de convertir un nombre en chaîne de caractères.

8 Opérateurs de manipulation de bits

Java Ca existe… pour les curieux, voir le manuel java p. 138

9 Opérateur « if arithmétique » : « ? : »

C ++
Java

Cet opérateur permet d'éviter une instruction `if` dans certains cas..

Exemple : une fonction qui retourne le maximum de deux entiers

```
public int max (int a, int b) {
     if (a>b) {
          return a;
     }
     else {
          return b;
     }
}
```

donnera avec un "if arithmétique"

```
public int max (int a, int b) {
     return a > b ? a : b;
}
```

Syntaxe : *test* **?** *valeurRetournéeSiVrai* **:** *valeurRetournéeSiFaux*

10 Opérateur « instanceof »

Java

Cet opérateur permet de contrôler le type d'un objet.

Exemple : `if (unObjet instanceof X) …`

Cet opérateur retourne le booléen `true` si l'objet `unObjet` est de type `X`.

En l'occurrence, il retournera `true` dans les cas suivants :

1/ si `unObjet` est une instance de `X` (créé par `new X()`)

2/ si `unObjet` est une instance d'une **sous-classe** de `X` (héritage)

3/ si `unObjet` est une instance d'une classe qui **implémente** `X`, au cas où `X` est une **interface**

11 Instruction composite : le « bloc »

C ++
Java

Ce mécanisme permet de fabriquer **une** instruction à partir d'une **séquence** d'instructions, au moyen d'une paire d'accolades.

```
{
     instruction ;
     instruction ;
     … ;
     instruction ;
}
```

Ceci constitue **une** seule instruction

Par exemple, une procédure est constituée d'une en-tête et d'un **bloc** :

```
public void afficherX() {
     System.out.println (x) ;
}
```

Syntaxe d'une procédure : *en-tête uneInstruction*

Autre exemple: un **bloc** pour écrire des `if` qui contiennent plusieurs instructions.

Exemple

```
if (x > 3) x=3;
```

Identique à

```
if (x > 3) {
     x = 3;
     autres instructions possibles..
}
```

Syntaxe du `if` : `if (`*test*`)` *uneInstruction*

12 La règle du point-virgule

C ++
Java

Le point-virgule doit obligatoirement terminer chacune des instructions.

Une exception.. il n'est pas nécessaire de terminer les blocs par un point-virgule (l'accolade suffit !)

13 L'instruction « if »

C ++
Java

Première forme →

```
if  (test) uneInstruction
```

Deuxième forme →

```
if  (test)
     uneInstruction
else
     uneAutreInstruction
```

- le test doit être écrit dans une paire de parenthèses
- ☺ il est fortement recommandé d'utiliser des **blocs** `{..}` pour écrire les instructions du `if`, même si les blocs ne contiennent qu'une seule instruction

Exemple no1	Exemple no2

`int x, y1, y2 ;`
`x=1 ;`
`if (x==0)`
`  y1 = x;`
`  y2 = x;`
`x = y2;`
 	
que vaut x ? *réponse : 1*
	`int x, y1, y2 ;`
	`x=1 ;`
	`if (x==0) {`
	`  y1 = x;`
	`  y2 = x;`
	`}`
	`x = y2;`
	
	que vaut x ? *réponse : 0*

Java Le test doit retourner un booléen : c'est vérifié par le compilateur

Note C++ En C++, le type booléen n'existe pas !

Ce qui est égal à `0` est considéré comme faux, tout entier différent de `0` est considéré comme vrai ..

Ainsi, exécutez (dans votre tête) le programme C++ suivant.. que vaut `x` pour finir ?

```
int x = 3 ;
if (x=4) x++ ;
→   x ?   réponse : 5
```

Essayez en Java, ce code sera refusé par le compilateur !

14 La pseudo clause « else if »

C ++
Java

La clause « `elsif` » n'existe pas, ni en C++, ni en Java.

Mais il est possible de simuler cette clause..

En effet, la syntaxe de l'instruction `if` permet d'écrire la chose suivante :

```
if (x==0) {
     action1 ;
}
else if (x==1) {
          action2;
     }
     else if (x==3) {
               action3;
          }
          else {
               action4;
          }
```

Voici le même programme écrit avec une autre indentation :

```
if (x==0) {
     action1 ;
}
else if (x==1) {
     action2;
}
else if (x==3) {
     action3;
}
else {
     action4;
}
```

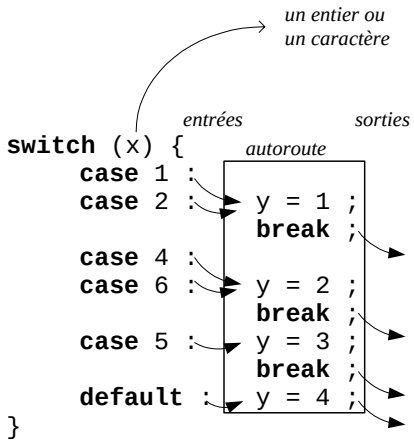
15 L'instruction « switch »

C ++
Java

Supposons l'instruction suivante en Ada..

```
case x is
     when 1..2      => y := 1;
     when 4, 6      => y := 2;
     when 5         => y := 3;
     when others    => y := 4;
end case;
```

La même chose en Java, C++

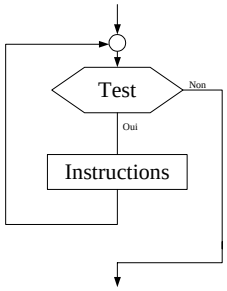


Les instructions du switch sont exécutées en séquence. Comme pour une autoroute, il y a plusieurs entrées possibles, et plusieurs sorties possibles.. Pour les adaistes, attention à ne pas louper la sortie !

16 Les boucles TANTQUE et REPETER

C ++
Java

```
TANTQUE Test FAIRE
    Instructions
FIN_TANTQUE
```

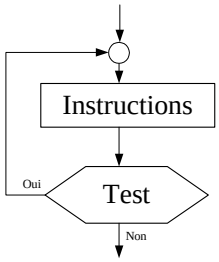



```
while(test)
   uneInstruction;
```

ou (avec un bloc)

```
while(test) {
     instruction 1;
     instruction 2;
     ..
}
```

```
REPETER
    Instructions
TANTQUE Test
```




```
do
     uneInstruction;
while (test);
```

ou (avec un bloc)

```
do{
     instruction 1;
     instruction 2;
     ..
}
while(test);
```

Un exemple

Voir le point suivant ([Boucle “`for`”]).

17 La boucle «`for`»

C ++
Java

La boucle `for` est un `while` déguisé..

Syntaxe

```
for (initialisation ; condition_d'itération ; enchaînement)
     uneInstruction
```

équivalent à

```
initialisation ;
while(condition_d'itération) {
     uneInstruction;
     enchaînement ;
}
```

Exemple : afficher l'alphabet à l'envers

```
for (char c = 'Z'; c >= 'A'; c--) {
     System.out.print (c);
}
```

Même chose avec un «while»

```
char c = 'Z';
while (c >= 'A') {
     System.out.print (c);
     c--;
}
```

Encore une dernière fois avec «do»

```
char c = 'Z';
do {
     System.out.print (c);
     c--;
}
while (c >= 'A');
```

Java A la différence de C++, - et de manière identique à Ada -, si la boucle `for` contient dans sa partie d'initialisation une déclaration de variable, il s'agit d'une **déclaration locale** à la boucle : la variable n'existe plus après exécution de la boucle, sa portée est limitée à la boucle.

18 Contrôler l'exécution d'une boucle: `break` et `continue`

C ++
Java L'instruction **`break`** termine une instruction itérative (l'équivalent du « `exit` » de Ada):

```
while (true) {
          ..
          ..
          if (x > 0) break ;
          ..
}
```

L'instruction **`continue`** termine l'itération en cours, mais la boucle continue de s'exécuter avec l'itération suivante.

```
while (! listeCaracteres.finListe()) {
        ch = liste.nextElement() ;
        if (ch != ' ') continue;     // ignorer
                                     // le caractère
        .. ;
}
```

19 Les types de données: objets et primitifs

Java

Java distingue deux types de données:

19.1 PRIMITIFS

Le nom commence par une **minuscule**.

- **entiers**

  `int` (32 bits), `long` (64 bits), `short` (16 bits) et `byte`(8 bits).

  Par défaut: `int`[1]

- **réels**

  `float` et `double` (double précision, x chiffres significatifs)

  Par défaut: `double`[2]

- **caractères**

  `char` (16 bits, selon le code UNICODE, qui comprend au départ le code ASCII)

- **booléens**

  `boolean` (8 bits)

1. Si on écrit «`3`», ce 3 est considéré comme une entité de type «`int`».
2. Si on écrit «`3.0`», ce `3.0` est considéré comme une entité de type «`double`».

19.2 OBJETS

Instances de classes dont le nom commence par une **majuscule**.

```
tableaux
String
Frame
Button
```

etc..

19.2.1 Manipulation par valeur, manipulation par référence

☝ **Les primitifs sont manipulés par valeur, et les objets par référence (pointeurs) !**

Exemple

```
int x = 3;
int y;
y = x;
y = 4;
```

Que vaut `x` ? .. toujours `3` !

Les primitifs sont en effet **manipulés par valeur**. L'affectation d'un primitif opère une **copie de valeur** : l'affectation `y=x` copie la valeur de `x` dans `y`.

Le même exemple avec des objets..

```
Button b2 = new Button ("OK");
Button b1;     // Déclaration d'une variable
b1 = b2;       // Affectation de la variable
b1.setLabel("Cancel");
```

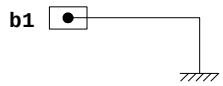
Quel est le label du bouton `b2` ? ⇨ "`Cancel`". En modifiant `b1`, `b2` a été modifié par la même occasion..

POURQUOI ?

Au contraire des primitifs, les objets sont **manipulés par référence**: les objets sont toujours créés dynamiquement, et les **variables** qui les désignent ne sont que des **pointeurs** sur les objets.

- Lorsque l'on déclare une variable de type objet, la variable est initialisée avec la valeur **`null`** (pointeur qui pointe sur rien du tout);

Button b1 ;

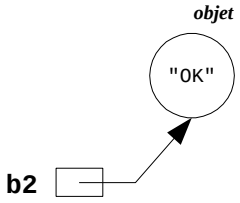


- Le programmeur a l'obligation de créer lui-même l'objet au moyen de l'opérateur **new;**

```
Button b2 = new Button ("OK");//Déclaration + création
```

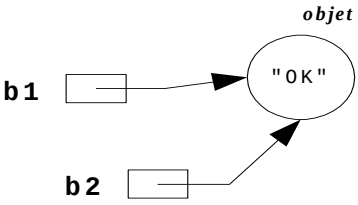
ou

```
Button b2;                          // Déclaration
b2 = new Button("OK");              // Création
```



- Le même objet peut être pointé par plusieurs variables, et peut être manipulé au travers de l'une ou l'autre des variables.

```
b1 = b2;
```




Au contraire des primitifs, l'affectation opère un **partage d'objets** plutôt qu'une copie de valeur.

☺ Comment opérer une copie d'objets ? ⇨ voir le point 46, page 64

- Le programmeur n'a pas besoin de détruire l'objet: il sera détruit automatique-

ment par le « garbage collector » de la machine virtuelle dès lors que cet objet ne sera plus pointé par une variable quelconque.

Exemple

```
public void p() {        // Une procédure..
     Button b = new Button("OK"); // Variable locale
                                                à la procédure
          :
          :
}     ⇨ Fin d'exécution de la procédure
⇨ La variable locale b est détruite
⇨ L'objet sera détruit par le « garbage collecteur » (ramasse-miettes)
```

19.2.2 Valeur d'initialisation

Contrairement aux langages Ada ou C++, les primitifs et les objets reçoivent une valeur d'initialisation:

- Les primitifs sont initialisés avec la valeur `0`

  entiers ⇨`0`

  réels ⇨`0.0`

  booléens ⇨`false`

  caractères ⇨caractère « nul » (dont le code UNICODE vaut `0`)

- Les variables de type objet sont des pointeurs initialisés avec la valeur **`null`**

---

20 Cycle de vie d'un objet

Considérons par exemple l'instruction: `unObjet = new Xxx() ;`

La ***création*** de l'objet `unObjet` comprend 3 étapes :

1/ allocation d'une zone mémoire

Afin de contenir les variables d'instance de l'objet. Ces variables d'instances sont initialisées avec leurs valeurs par défaut ou par la valeur d'initialisation spécifiée par le programmeur lors de la déclaration de la variable.

Par exemple :

```
int i ;       // "i" est initialisée à 0 (par défaut)
int j = 3 ;   // "j" est initialisée avec 3
```

2/ invocation du constructeur (`Xxx()`)

Lors de cette opération, de nouvelles valeurs peuvent être affectées aux variables d'instance et aux variables de classe. Voir aussi le point [Les constructeurs].

3/ retour de la référence

L'opérateur **`new`** retourne pour finir la référence à l'objet, qui sera affectée dans notre cas à la variable `unObjet`.

La ***destruction*** de l'objet est opérée par le ramasse-miettes (garbage collector). Cette opération est susceptible d'arriver dès l'instant où l'objet n'est plus référencé par aucune variable.

Avant même de détruire l'objet, le garbage collector envoie le message **`finalize()`** à cet objet, lui donnant ainsi l'occasion d'accomplir ses dernières volontés. L'utilisation de **`finalize`**(), qui joue le même rôle que le destructeur de C++, est effectuée très rarement en Java, du fait de l'existence même du garbage collector.

Le programmeur peut la redéfinir ainsi :

```
public void finalize () {
     .../* dernières volontés ..*/
}
```

Voir aussi le point [Les destructeurs].

21 Les tableaux

C ++
Java

Les tableaux de Java sont avant toute chose des objets. A ce titre, ils jouissent des particularités propres aux objets:

- ils sont manipulés par référence,
- ils sont alloués dynamiquement au moyen de l'opérateur **`new`**,
- ils sont récupérés automatiquement par le « ramasse-miettes » (le « garbage collector ») quand ils ne sont plus référencés,
- ils héritent de la classe «`Object`».

Toutefois, à la différence des « vrais » objets, il n'existe pas de classe correspondante (la classe `Array` n'existe pas !).

21.1 DÉCLARATION D'UN TABLEAU

```
int tabEntiers[];  // en utilisant la syntaxe de C/C++
                   // ou,
int [] tabEntiers; // en utilisant une syntaxe propre à Java
```

```
int [][] deuxDimensions;          // tableau à deux dimensions
int  troisDimensions [][][];      // tableau à trois dimensions
```

Les 3 tableaux déclarés ci-dessus sont de simples références initialisées à «**null**». Les tableaux eux-mêmes (des objets) ne sont pas encore créés. C'est au moment de leur allocation dynamique que l'on pourra spécifier leur taille.

21.2 CRÉATION DYNAMIQUE DES OBJETS « TABLEAU »

La création d'un objet de type tableau peut être réalisée de deux manières:

- Avec l'opérateur **new**, en indiquant le nombre de cases du tableau

```
int [] tabEntiers;           // Déclaration
tabEntiers = new int[4];     // Création: 4 entiers
                             // initialisés à 0

                             // OU,

int [] tabEntiers = new int[4];  // en une seule
                                 // instruction..
```

  Les éléments d'un tableau reçoivent une valeur initiale qui correspond à la valeur par défaut du type des éléments du tableau : («**0**» pour un tableau d'entiers, «**null**» pour des objets ou des tableaux, ..

- Au moyen d'un « **initialiseur** ».

```
int x = 4 ;
int [] tabEntiers = {1, x, 2*x, 4};
// Tableau de 4 entiers, initialisés à 1, 4, 8 et 4
```

  L'initialiseur doit être utilisé au moment de la déclaration. Il est interdit dans le cadre d'une affectation. L'instruction ci-après est illégale.

```
int [] tab ;
tab = {1, 2} ;          // Illégal !!
```

21.3 ACCÈS AUX ÉLÉMENTS D'UN TABLEAU

Les tableaux sont indexés par des valeurs de type «`int`», en commençant à «**0**». Des indices de type «`long`» ne sont pas autorisés. La taille des tableaux est donc limitée à 32 bits !

```
tabEntiers [3] = 4;
System.out.println (tabEntiers [0]);
```

☞A l'exécution, si l'indice sort des limites du tableau, le noyau Java lève l'exception : «`ArrayIndexOutOfBoundsException`».

21.4 LA TAILLE D'UN TABLEAU

Les tableaux sont des objets, et ils disposent d'un attribut (un seul) qui caractérise leur taille : l'attribut **`length`**.

Voici un fragment de programme qui initialise tous les éléments du tableau `tab` avec la valeur **`2`** :

```
for (int cpt = 0 ; cpt < tab.length ; cpt++) {
     tab [cpt] = 2 ;
}
```

☞Attention ! rappelons que l'indice d'un tableau commence par «0».

Ainsi, l'instruction «`tab[tab.length]=3;`» lèvera l'exception «`ArrayIndex-OutOfBoundsException`»

☞La taille d'un tableau est statique[1]: le nombre des éléments ne peut pas être augmenté ni diminué dynamiquement, au grès des besoins.

21.5 UN TABLEAU COMME ARGUMENT DE MÉTHODE

Un paramètre formel de type tableau sera spécifié en utilisant la même syntaxe que pour la déclaration d'un tableau :

```
void uneProcédure (int [] tabEntiers) {
// Cette procédure diminue de 1 tous les éléments
// du tableau
     for(int cpt=0; cpt < tabEntiers.length ; cpt++) {
          tabEntiers [cpt] --;
     }
}
```

21.6 UN TABLEAU COMME TYPE DE RETOUR

Une fonction peut retourner un tableau. Voici par exemple l'entête d'une telle fonction :

```
int [] uneFonction () {
// Fonction retournant un tableau d'entiers
          :
```

1. La classe « `Vector` » du paquetage « `java.util` » met en œuvre des listes dynamiques et permet ainsi de manipuler des tableaux de taille variable

```
        return unTableauDEntiers ;
    }
```

21.7 TABLEAUX À DIMENSIONS MULTIPLES

Les « initialiseurs » de tableaux peuvent être imbriqués :

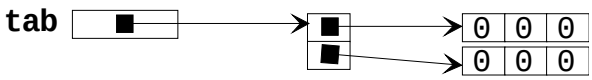
```
int [][] deuxDimensions = { {1, 2}, {3, 4}, {5, 6} } ;
```

équivalent à :

```
int [][] deuxDimensions = new int [3][2] ;
deuxDimensions [0][0] = 1 ;
deuxDimensions [0][1] = 2 ;
deuxDimensions [1][0] = 3 ;
...
deuxDimensions [2][1] = 6 ;
```

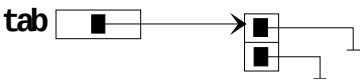
- La création d'un tableau à dimensions multiples peut être opérée en une étape

```
int [][] tab = new int [2][3] ;
```

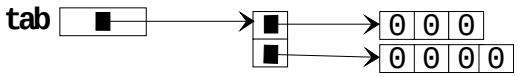


- ou en plusieurs étapes :

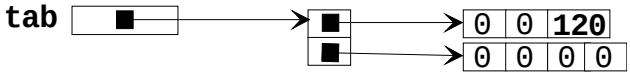
```
int [][] tab = new int [2][] ;
    // Création d'un tableau de deux éléments dont le type
    // est "int []", initialisés à "null" pour l'instant
```



```
tab[0] = new int [3] ;
tab[1] = new int [4] ;
      // On remarque dans cet exemple que ce tableau à deux
      // dimensions n'est pas une matrice carrée ! !
```




```
tab [0][2] = 120 ;
      // Accès à un élément du tableau à deux dimensions
      // Le raccourcit "tab [0, 2]" à la "pascal" n'est pas
      // prévu dans Java.
```



22 Les chaînes de caractères (`String`)

En Java, les chaînes de caractères ne sont pas des tableaux de caractères !

En Java, on sait simplement que les chaînes sont des objets, des instances de la classe «`java.lang.String` ».

En tant que chaînes, ces objets jouissent toutefois de deux particularités :

- Il est possible de les créer au moyen d'un initialiseur (texte écrit entre guillemets), sans passer par l'opérateur «`new`».

  ```
  String txt = "Salut";
  ```

- Java définit un opérateur de concaténation: l'opérateur «+»

  ```
  System.out.println ("2 + 3 = " + (2+3) + " !");
  ```

Dans une expression qui comprend « l'addition » de plusieurs opérandes, il suffit que l'un d'entre eux soit de type `String`, pour que les autres opérandes soient automatiquement convertis en `String` (si cette conversion est possible).

Voici quelque méthodes proposées par la classe `String`:

Soit la chaîne «s» : `String s = "Salut ";`

- `s.length()`

  Retourne le nombre de caractères de «s»

- `s.charAt(1)`

  Retourne «a», le caractère de rang «1». Le rang d'un caractère va de 0 à (`s.length()-1`)

- `s.equals(uneAutreChaîne)`

  Retourne «`true`» si les deux chaînes ont la même valeur

- `s.compareTo(uneAutreChaîne)`

  Retourne «`-1`» (si plus petite que..), «`0`» ou «`1`»

- `s.indexOf (uneSousChaîne, depuisUnIndex)`

  Retourne la position de la sous-chaîne

- `s.concat (uneAutreChaîne)`

  Retourne une chaîne résultant de la concaténation avec une autre chaîne

- `s.trim ()`

  Retourne une chaîne sans les espaces de début et de fin

- `s.toUpperCase ()`

  Retourne une chaîne transformée en majuscules

 Pour comparer le contenu de deux chaînes de caractères, utiliser la fonction equals : `if (s1.equals(s2))`…

Une simple comparaison ne marchera pas : `if (s1==s2) ..` (comparaison de pointeurs !!)

 Une chaîne de caractères est un objet non mutable : son contenu est défini une fois pour toutes par l'initialiseur.

23 Les articles (« records »)

C++
Java

Un article sera défini au moyen d'une classe.

Fondamentalement, une classe est un type article **amélioré**, offrant en plus la possibilité de rassembler la définition de la structure de données avec les méthodes de manipulation de la dite structure.

Comme exemple, considérons la définition d'un nombre complexe.

En Ada

```
record TComplexe is
    re: float;
    im: float;
end record;
```

```
c : TComplexe ;
begin
    c.re := 3.5 ;
    etc..
```

En Java

```
class Complexe {
  public double re;
  public double im;
}
```

```
Complexe c ;
c.re = 3.5 ;
etc..
```

En C++

```
class Complexe {
   public :
      double re;
      double im;
}
   ou
struct Complexe {
   double re, im;
}
```

```
Complexe c ;
c.re = 3.5 ;
etc..
```

24 Le type intervalle

Java

Désolé, mais les types définis par le programmeur sont limités aux tableaux et aux classes. Le type Intervalle n'existe pas non plus en C++.

En conséquence de quoi, si on tient absolument au type intervalle, il est toujours possible de créer une classe correspondante.

Par exemple, pour un intervalle d'entiers, on peut déclarer la classe suivante:

```
class MonIntervalle {
     private int min = 100 ;
     private int max = 200 ;
     private int valeur ;

     public void setValeur(int val) {
          if (val < min ¦¦ val > max)
                    throw new RunTimeException();
          valeur = val;
     }
     etc..
}
```

Mais, évidemment, c'est lourdingue..

25 Le type énuméré

Java

Désolé, mais les types définis par le programmeur sont limités aux tableaux et aux classes.

Pourtant, C++ connaît les types énumérés.. Ici, Java est victime de sa volonté de simplification.

Comment simuler un type énuméré en Java ?

⇨ en utilisant les concepts de classe et de COPIER-COLLER, et en remarquant que la déclaration d'un type énuméré revient à déclarer un ensemble de constantes globales.

Dans l'exemple qui suit, le mot-clé **`final`** indique qu'il s'agit de constantes (voir le point 45, page 64) et le mot-clé **`static`** indique qu'il s'agit de variables de classe (voir le point 44, page 60).

```
class Semaine {
     public static final Samedi = 0;
     public static final Dimanche = 1;
     public static final Lundi = 2;
     etc..
}
```

Utilisation

```
int jour = Semaine.Samedi ;
jour += 1 ;// jour == Dimanche ;
if (jour >= Semaine.Lundi && jour <= Semaine.Vendredi)
{
     // Si c'est un jour de semaine..
```

```
        ...
    }
```

Mais, évidemment, c'est lourdingue..

26 Conversions de types

26.1 PRIMITIFS ⇔ PRIMITIFS

Toutes les conversions sont autorisées, sauf celles qui font intervenir le type `boolean`.

Les conversions qui entraînent un **rétrécissement** comme par exemple :

- une perte de précision: « réel ⇨ entier »
- une réduction du nombre de bits: « `int` (32 bits) ⇨ `short` (16 bits)», etc..

réclament **obligatoirement** une **coercition de type**: `(type)`. La coercition de type est une espèce d'avertissement au programmeur : elle n'empêchera pas le programme de « planter » à l'exécution le cas échéant, voir ci-après.

Les conversions sans risque, celles qui n'entraînent pas de rétrécissement, sont opérées automatiquement.

- entier ⇔ entier

```
int i = s;      // Avec "s" de type short
                // c pas de rétrécissement

short s = (short) i;// Réduction du nombre de bits !
                    // Plante à l'exécution si "i" utilise plus de 16 bits
etc..
```

- réel ⇔ entier

```
double d = i;       // Avec "i" de type entier pas de rétrécissement

int i = (int)d;     // Rétrécissement avec troncation des chiffres
                    // après la virgule
```

- int (32 bits) ⇔ char (16 bits unicode)

```
int i = 'A';                // Unicode de 'A' (65)  affecté à "i"
```

```
char c = (char)65;        // Valeur 'A' affectée à "i"
                          // Coercition obligatoire car réduction
                          // du nombre de bits !
```

- etc..

26.2 PRIMITIFS NUMÉRIQUES ⇔ STRINGS

- ⇨ Strings

  `String s = unNumérique + "";` *Concaténation avec un texte vide*

  ou: `String s = String.valueOf(unNumérique);`

- ⇨ Numériques

```
int i    = (new Integer(unString)).intValue();
double d = (new Double(unString)).doubleValue();
float f  = (new Float(unString)).floatValue();
etc..
```

26.3 CHAR ⇔ STRING

```
String s = new String (c);  // avec "c" de type char
char c = s.charAt(indice);  // avec "s" de type String
```

26.4 BOOLEAN ⇔ STRING

```
boolean b = Boolean.getBoolean (unString) ;
String s = String.valueOf(unBooléen);
```

26.5 PRIMITIFS ⇔ OBJET

Cette conversion utilise les classes "wrappers" spécifiques, qui permettent d'encapsuler le primitif correspondant.

primitif	*classe wrapper correspondante*
int	Integer
double	Double
char	Character

boolean	Boolean
byte	Byte
short	Short
long	Long

- Primitif ⇨ Wrapper

```
Integer ii = new Integer (i);   // avec i de type int
Double dd  = new Double (d);    // avec d de type double
etc..
```

- Wrapper ⇨ Primitif

```
int i = ii.intValue();          // avec ii de type Integer
double d = dd.doubleValue();    // avec dd de type Double
etc..
```

26.6 OBJET ⇔ OBJET

La conversion Objet ⇔ Objet n'existe pas au niveau du langage. Si le programmeur éprouve le besoin d'opérer de telles conversions, c'est à lui de les implémenter ! Ce cas n'arrive de toute manière que très rarement et le mode de conversion dépend toujours du problème et des types d'objets à convertir l'un vers l'autre.

Lors d'une affectation, le compilateur peut exiger du programmeur qu'il opère explicitement une **coercition de type** !

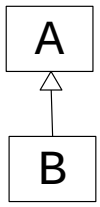
Par exemple:

- soit `a` une variable désignant un objet de type `A` (type de déclaration)
- soit `b` une variable désignant un objet de type `B` (type de déclaration)

☺ L'affectation `a = b` sera acceptée par le compilateur si tous les messages du type `A`, susceptibles d'être envoyés à `a`, sont prévus également au niveau du type `B`.

Autrement dit, cette affectation sera acceptée :

- si le type `B` est identique au type `A`,
- si le type `B` est une **sous-classe** de `A` (héritage)

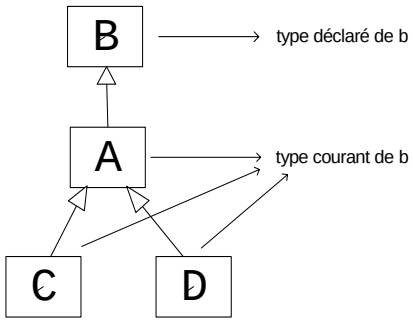


- ou encore si le type B **implémente** A (si A est une interface)

Au contraire, si B est une **super-classe** de A (héritage), l'affectation sera acceptée si le programmeur écrit une coercition de type:

```
a = (A)b;
```

Toutefois, cette affectation ne passera à l'exécution (sans lever d'exception) que si le **type courant** de l'objet b est de type A ou d'un type dérivé de A par héritage.



27 Procédures, fonctions et envoi de message

C ++
Java

Les procédures et fonctions sont des regroupements d'instructions sous le même nom. En POO, les fonctions et procédures sont regroupées sous un seul terme : on parle de **méthodes**.

Pour exécuter les instructions d'une méthode, il nous faut **invoquer** cette dernière en envoyant le **message** correspondant à l'objet sur lequel opère la méthode:

```
objet.message(paramètres)
```

Pour envoyer un message, il faut :

1/ écrire le nom de l'objet,
2/ écrire un point,
3/ écrire le nom de la méthode, en écrivant les paramètres entre parenthèses.

En C++ comme en Java, les parenthèses sont obligatoires, même s'il n'y a pas de paramètre à écrire.

Exemples

```
unRectangle.getLargeur() ;
unePile.vider()  …
```

28 Pour s'envoyer un message: «this»

Si on omet d'écrire le nom de l'objet, cela signifie que le mot «this» est sous-entendu : le message est envoyé à « lui-même » (un message est toujours envoyé **par un objet** à un objet).

Exemple:

```
vider();
```

équivalent à:

```
this.vider();
```

29 Méthodes d'instance et méthodes de classe

Java

Dans la majorité des cas, les messages sont envoyés à des objets (des **instances de classes**).

Certains messages seront toutefois envoyés à **la classe elle-même** directement.

Ces dernières méthodes sont appelées **méthodes de classe**, les autres étant désignées par le terme **méthodes d'instance.** La déclaration d'une méthode de classe comporte le mot-clé **static**.

Un premier exemple : la fonction « main » dans un programme avec 2 classes

```
class Y {
     public void yP1() {
```

```
            System.out.println("yP1:je suis une methode d'instance ");
            yP2();// Correct
      }
      public static void yP2() {
            System.out.println("yP2:je suis une methode de classe");
            yP1();// REFUSE ! (voir ci-après)
      }
}
public class X {
      public void xP1() {
            System.out.println("xP1:je suis une methode d'instance ");
            xP2();// Correct
      }
      public static void xP2 () {
            System.out.println("xP2:je suis une methode de classe");
            xP1();// REFUSE !(voir ci-après)
      }


      public static void main (String [] arg) {
      // "main" est une méthode de classe
            Y unObjet = new Y();
            unObjet.yP2();// Invocation d'une méthode d'instance
            Y.yP1();      // Invocation d'une méthode de classe
            xP1();    // REFUSE ! (voir ci-après)
            xP2();    // Correct
      }
}
```

Invocation des méthodes d'une classe depuis l'extérieur de la classe

- Invocation d'une méthode d'instance (yP1)

```
Y uneInstance = new Y();
uneInstance.yP1();
```

- Invocation d'une de méthode de classe (yP2)

```
Y.yP2();
```

Par exemple, au lancement du programme, la machine virtuelle envoie le message `main` directement à la classe X: **`X.`**`main(`*`arguments`*`)`

Invocation des méthodes d'une classe depuis l'intérieur de la classe

☺ Une méthode d'instance peut invoquer des méthodes d'instance ou des méthodes de classe

☹ Une méthode de classe ne peut invoquer **que des méthodes de classe !**

POURQUOI CETTE LIMITATION ? à ce stade, cela peut paraître curieux..

Ainsi, la fonction **`main`** ne peut pas invoquer de méthode d'instance !

➪ nous reprendrons la question dans le point [Variables de classes et variables d'instance].

Un deuxième exemple : la fonction « sin »

Les fonctions mathématiques usuelles sont rassemblées dans la classe « `java.lang.Math` ».

Pour faciliter leur utilisation, elles sont proposées sous la forme de méthodes de classes:

```
class Math {
     public static double sin(double x) {...}
     public static double cos(double x) {...}
     etc..
}
```

Ainsi, pour utiliser la fonction `sin`, il est inutile de créer une instance de la classe `Math` ! On écrira par exemple :

```
double y = Math.sin(x);
```

---

30 Déclaration d'une méthode

C ++
Java

30.1 DÉCLARATION D'UNE FONCTION

En ada

```
function Somme (n1, n2 : integer) return integer is
     begin
          return n1 + n2 ;
     end Somme ;
```

En Java, C++

```
int somme (int n1, int n2) {
     return n1 + n2 ;
}
```

L'entête de la fonction commence par indiquer le type de la valeur qui sera retournée par la fonction (dans notre exemple, il s'agit du type «`int`»).

L'instruction `return` est **obligatoire**. Elle termine l'exécution de la fonction.

30.2 DÉCLARATION D'UNE PROCÉDURE

En ada

```
procedure AfficherTableau (tab: array of integer) is
     begin
          for  cpt in 1..tab'length loop
               put (tab(cpt));
          end loop;
     end AfficherTableau ;
```

En Java, C++

```
void afficherTableau (int [] tab) {
     for (int cpt = 0 ; cpt < tab.length ; cpt++)
          System.out.print (tab[cpt]);
}
```

Les procédures sont en réalité de simples fonctions qui ne retournent rien (c'est à dire « **`void`** »)

En principe, une procédure ne contient pas d'instruction «**`return`**»

L'instruction « **`return ;`** », utilisée dans une procédure, a pour effet de terminer l'exécution de la procédure. On opère très rarement ainsi.

---

31 Invoquer une fonction

C ++
Java

On invoque une fonction pour interroger un objet : l'objet qui reçoit le message opère un calcul et répond en retournant la valeur du résultat.

☞ En principe, l'objet que l'on interroge **ne change pas de valeur !!**

☺ Le **nom** donné à une fonction est celui d'un **nom commun ou d'un adjectif**, qui caractérise la nature de la valeur retournée.

Nous adopterons les conventions suivantes :

- Les noms des fonctions qui retournent la valeur d'un attribut de l'objet seront nommées « **get***NomAttribut*() » :

  getSommet(), **get**Largeur(), **get**Maximum(),……

- Les noms des fonctions qui testent l'état d'un objet en retournant un booléen seront nommées « **is***NomEtat* », ou « **est***NomEtat* » en français : **est**Rouge(), **est**Vide(), **est**EnMarche(), …

☞ Une invocation de fonction **est une valeur**, que l'on peut utiliser au sein d'une **expression** :

```
int x = 3 + unRectangle.getLargeur() ;
if (unePile.estVide()) {..}
```

☞ Dans la majorité des cas, le fait de ne pas utiliser la valeur retournée par une fonction n'a pas de sens, même si c'est accepté par le compilateur :

```
unRectangle.getLargeur() ;    // cette instruction n'a pas de sens !
```

32 Invoquer une procédure

C ++
Java

☞ **On invoque une procédure pour donner un ordre à l'objet, lui demander d'accomplir une action** : l'objet accomplit l'action et ne retourne aucune valeur.

☞ L'objet à qui l'on demande d'accomplir une action **peut changer de valeur** !! (différence avec les fonctions).

☺ Le **nom** donné à une procédure est celui d'un **verbe à l'infinitif**, qui dénote une action.

Nous adopterons les conventions suivantes :

- Les noms des procédures utilisées pour mettre à jour la valeur d'un attribut de l'objet seront nommées « **set***NomAttribut*» :

```
setLargeur(15), setCouleur(Color.blue), ..
```

- Les autres, qui demandent à l'objet d'accomplir une action, seront simplement nommées au moyen d'un verbe à l'infinitif :

```
empiler(..), dessiner(),  ajouter(..),..
```

Une invocation de procédure **constitue une instruction du programme** à part entière (au contraire des fonctions):

```
unDessin.afficher () ;
unePile.empiler (4) ;
unRectangle.deplacer(n1, n2) ;
```

33 Mécanisme de passage des paramètres

Java En C++, le programmeur a le choix entre un mécanisme de passage par valeur et un mécanisme de passage par référence. En Java, il n'y a pas le choix..

33.1 Pour les primitifs, il s'agit d'un passage par valeur (mode « in »)

Donc, la valeur du paramètre effectif est **constante** : elle ne sera jamais modifiée par l'invocation de la méthode.

Le paramètre formel est une variable qui contient une copie de la valeur de l'original (l'original étant le paramètre effectif).

Ainsi, même si la valeur du paramètre formel est modifié par les instructions de la procédure, la valeur du paramètre effectif reste intacte.

☞*Voir le paramètre formel « x » dans l'exemple ci-après*

33.2 Pour les objets, il s'agit d'un passage par référence (mode « in out »)

Le paramètre formel et le paramètre effectif sont des variables qui désignent **le même objet**.

Ainsi, si le paramètre effectif est un objet, sa valeur peut être modifiée par l'invocation de la méthode.

☞*Voir le paramètre formel « b1 » dans l'exemple ci- après,*

Par contre, la variable elle-même (c'est à dire la référence) est **passée par valeur** : si la méthode modifie cette référence (pour que le paramètre formel désigne un autre objet), la variable correspondant au paramètre effectif ne sera pas modifiée et désignera toujours le même objet.

☞*Voir le paramètre formel « b2 » dans l'exemple ci- après*

Exemple

```
import java.awt.*;
public class Essai {

     public static void uneMethode(int x, Button b1, Button b2)
     {
          x = x + 1;                      // Modification de la valeur
          b1.setLabel("Bye Bye");         // Modification de la valeur
          b2 = new Button ("Bye Bye") ;   // Modification de la référence
     }

     public static void main (String [] arg) {
          Button b1 = new Button ("Hello B1");
          Button b2 = new Button ("Hello B2");
          int valeur = 1;
          uneMethode(valeur, b1, b2);
          System.out.println ("Resultat = " +
                              valeur + "-" +
                              b1.getLabel() + "-" +
                              b2.getLabel()
                             );
          // Affiche: Resultat = 1-Bye Bye-Hello B2
     }
}
```

34 Imbrication des méthodes

C ++
Java

Désolé, mais la déclaration de méthodes à l'intérieur d'une méthode n'est pas prévu !

Ce mécanisme, que l'on trouve par exemple en Ada, est propre aux langages prévus pour la programmation procédurale afin d'encourager la décomposition fonctionnelle.

En POO, les méthodes sont en général très courtes et ne nécessitent pas de décomposition fonctionnelle.

En revanche, Java et C++ permettent la décomposition d'objet en proposant le concept de classe imbriquée (ou « classe interne »).

35 Récursivité

C++
Java

C'est possible en Java ou en C++ : une méthode peut s'appeler elle-même.

A titre d'exemple, la fameuse fonction « **factoriel** »

```
public static int factoriel (int n) {
     if (n > 1)
          return n*factoriel(n-1) ;
     else return 1 ;
}
```

36 Surcharge des méthodes

C++
Java

On peut donner **le même nom** à deux ou à plusieurs méthodes qui accomplissent en principe la même fonction logique.

Voici un exemple avec deux exemplaires de la fonction «Max»

```
class Mathematiques {

     public static void double Max (double a, double b)
     {
                    return a > b ? a : b ;
     }

     public static void int Max (int a, int b) {
                    return a > b ? a : b ;
     }
}
```

Utilisation de `Max`

```
double x = Mathematiques.Max(3.2, 4.5) ; //Première méthode
int x = Mathematiques.Max (3, 4) ; // Deuxième méthode
```

Où surcharger les méthodes ?

Il est possible de surcharger des méthodes, soit dans une même classe, soit dans une classe héritière qui surchargera alors des méthodes de ses classes parentes.

Condition à respecter : signature différente

La surcharge est effective dès lors qu'il y a modification des arguments de la méthode surchargée. Les différentes versions doivent avoir des ***signatures*** différentes.

Rappelons que la signature d'une méthode comprend son nom et ses paramètres, mais ne comprend pas le type retourné. Une distinction limitée au type seulement entraînerait des ambiguïtés.

36.1 SURCHAGE DES OPÉRATEURS

Java

Désolé, mais la surcharge des opérateurs n'est pas prévue en Java. Ce concept est pourtant proposé par C++ et Ada, et il est très pratique aux yeux de certains. Malgré tout, les auteurs de Java ne l'ont pas retenu en invoquant des raisons qui tiennent du « Génie Logiciel » : la surcharge des opérateurs diminue la lisibilité des programmes.

36.2 LA SURCHARGE DES MÉTHODES EST UN VILAIN DÉFAUT

Java

☺ Pour assurer une certaine lisibilité des programmes, la surcharge des opérateurs n'est pas autorisée en Java. Au même titre, les auteurs de Java recommandent vivement de ne pas abuser de la surcharge des méthodes . D'ailleurs, les lecteurs familiers des librairies du JDK auront certainement remarqué que la surcharge des méthodes y est rarement utilisée.

Ainsi, les classes dédiées aux entéres/sorties comme `DataOutputStream`, `ObjectOutput` , etc.. n'hésitent pas à donner des noms différents aux différentes procédures d'écriture : `writeObject`, `writeInt`, `writeBool`, `writeChar`, etc.. alors que le nom `write` aurait pu être prévu pour toute ces procédures !

Toutefois, la surcharge est utilisée dans les librairies Java. C'est le cas par exemple des constructeurs..

Pour mieux comprendre la philosophie de Java, remarquons que le langage, - à l'instar de C++ -, ne connaît pas la notion de **paramètres avec valeur par défaut**. Ce concept

est proposé par Ada. Encore une fois, Java s'appuie sur la lisibilité pour justifier son choix : plutôt que de prévoir une seule procédure dont les paramètres ont des valeurs par défaut, Java préfère utiliser le concept de surcharge et déclarer une collection de méthodes qui se distinguent par le nombre de leur paramètres, suivant que certains ont ou non des valeurs par défaut.

Ce principe est utilisé avec les constructeurs !

A titre d'exemple, citons les différents constructeurs du gestionnaire de disposition `FlowLayout` :

- FlowLayout()
  Constructs a new Flow Layout with a centered alignment and a default 5-unit horizontal and vertical gap.

- `FlowLayout(int align)`
  Constructs a new Flow Layout with the specified alignment and a default 5-unit horizontal and vertical gap.

- `FlowLayout(int align, int hgap, int vgap)`
  Creates a new flow layout manager with the indicated alignment and the indicated horizontal and vertical gaps.

37 Traitement des exceptions

C++
Java

Principe en gros identique en C++, mais avec une syntaxe différente

Le mécanisme des exceptions est prévu pour traiter les erreurs détectées pendant l'exécution du programme. Il permet en outre de distinguer clairement l'endroit du programme où l'erreur peut arriver de l'endroit du programme où l'erreur est traitée.

Avec le mécanisme des exceptions, le programmeur a la possibilité :

1/ de **traiter** l'erreur directement par la méthode dans laquelle l'erreur s'est produite, on dit que l'erreur est **récupérée**,

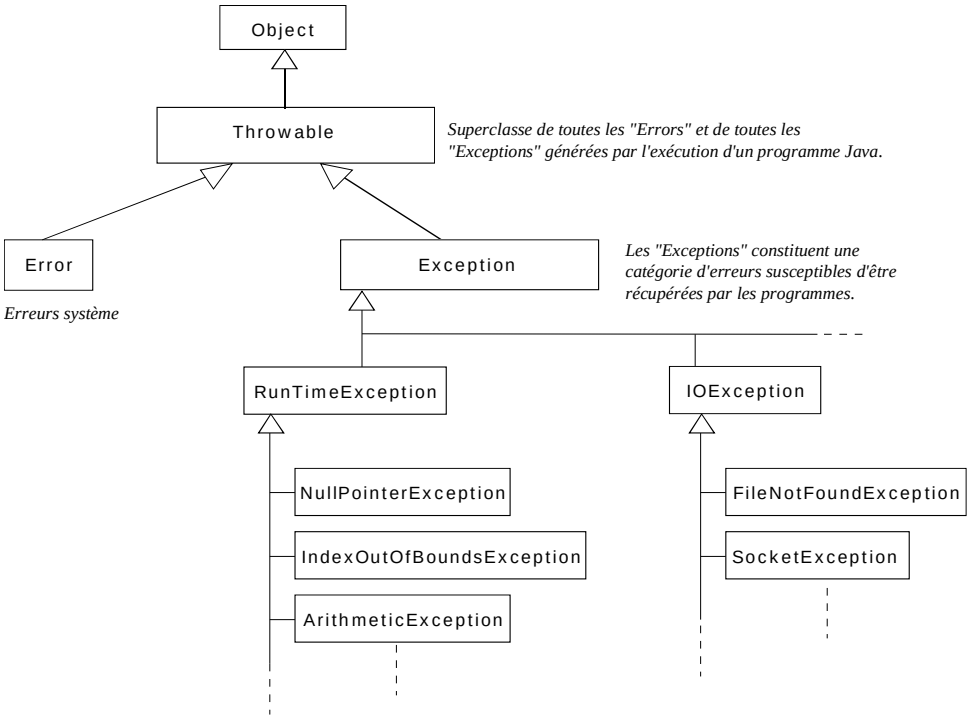
2/ ou alors, de **propager** l'erreur pour être traitée au niveau supérieur (la méthode appelante), et ainsi de suite. On utilise la propagation d'erreur au cas où la méthode dans laquelle l'erreur a été constatée n'est pas compétente pour faire son traitement. En remontant les étages, si aucune méthode du programmeur ne récupère l'exception, cette dernière est alors traitée auto-

matiquement et conduit en général à une erreur fatale

37.1 CLASSIFICATION DES EXCEPTIONS

Les exceptions sont des **objets**, instances de la classe `Exception` ou de classes dérivées de `Exception`. La classe `Exception` hérite elle-même de la classe générale «`Throwable`». De ce fait, toutes les exceptions sont caractérisées par un attribut de type «`String`», un texte qui spécifie le message d'erreur associée à l'exception. Ce message d'erreur peut être spécifié grâce au paramètre du constructeur de la classe «`Exception`».

Considérons la hiérarchie des classes d'exception offerte en standard par l'environnement Java :



La classe «**`Throwable`**» contient un autre attribut remarquable. Ce dernier contient une « photo » de la pile d'exécution au moment où l'erreur a été signalée. Cette pile d'exécution, - qui mémorise la file des appels de méthodes -, permet de déterminer les circonstances dans lesquelles l'erreur s'est produite. Si, - d'aventure -, l'application Java « plantait », cette pile d'exécution serait alors affichée automatiquement dans la fenêtre d'exécution.

La classe «**`Exception`**» définit l'ensembles des exceptions pouvant être gérées directement par le programmeur. De cette classe sont dérivées de nombreuses classes standard.

Citons notamment la classe «`IOException`» qui regroupe les erreurs d'entrées/sorties. Comme par exemple la classe «`SocketException`» qui représente les erreurs de connexion sur le réseau Internet.

Citons encore la classe «**`RunTimeException`**» qui regroupe les erreurs classiques de programmation, comme :

- «`NullPointerException`»
  Envoi de message à une référence «`null`»
- «`IndexOutOfBoundsException`»
  Accès à un tableau en dehors des limites
- «`ArithmeticException`»
  Division par zéro, dépassement de capacité d'un entier

37.2 PROPAGER UNE EXCEPTION

Considérons l'entête de la méthode **`setRayon`** :

```
public void setRayon (int valeur) throws RayonException {
          …
}
```

L'en-tête de cette méthode comporte une clause spéciale : «**`throws RayonException`»,** qui signale que l'exécution de cette méthode est susceptible de lever une exception.

La double utilité de la clause **`throws`**:

1/ Elle améliore la lisibilité du programme en indiquant que l'exécution de cette méthode est susceptible de lever une exception ;

2/ Elle indique au compilateur que toute méthode qui envoie le message `setRayon` doit traiter l'exception :

- soit en la **récupérant** au moyen d'un **`catch`** (voir plus loin),
- soit en la **propageant**, c'est-à-dire en renvoyant le traitement de l'exception à sa propre méthode appelante, comme par exemple le constructeur de la classe `Cercle` :

```
public setCercle (int x, int y,
                  int rayon) throws RayonException {
        setRayon(rayon) ;
        this.x = x ; this.y = y ;
```

```
}
```

Dans un tel cas, l'écriture de la clause **`throws`** est obligatoire : le compilateur le vérifie[1].

37.3 RÉCUPÉRER UNE EXCEPTION

L'instruction «`try`».. «`catch` » permet de récupérer la levée d'une exception en écrivant un code de traitement, comme par exemple :

```
void uneMéthode () {
        Cercle c ;Déclaration d'un objet de type Cercle
        int leRayon ;
        /*
        Bout de programme permettant de saisir les valeurs du rayon,
        de x et de y
        ..
        */
        try {On essaye ..
            Peut-être qu'une exception sera levée
            c = new Cercle (100, 100, leRayon) ;
            L'exception n'a pas été levée, le programme continue ici
            ..
        }
        catch (RayonException e){
            Une exception a été levée
            Le paramètre «e» est l'objet qui correspond à l'exception

            Par exemple, voici une utilisation de cet objet en affichant le
            message d'erreur associé à cette exception
            System.out.println(e.getMessage()) ;

            Ici, on écrit le traitement de l'exception
            ..
        }
        On arrive à cet endroit de la méthode qu'il y ait eu ou non une
        levée d'exception
}
```

37.4 DIFFÉRENCIER LES TYPES D'EXCEPTIONS (CASCADE DE « CATCH »)

Pour distinguer les exceptions et leur affecter un traitement particulier, il est possible d'écrire plusieurs clauses «`catch`», comme ci-dessous :

```
try {
        On essaye ..
        ..code pouvant lever une exception ..
```

1. Sauf pour les exceptions de type `RuntimeException`, voir plus loin

```
}
catch (NullPointerException e) {
        Traitement de l'erreur NullPointerException
}
catch (IndexOutOfBoundsException e) {
        Traitement de l'erreur IndexOutOfBoundsException
}
catch (IOException e){
        Traitement de l'erreur IOException
}
```

Le principe est le suivant : quand une exception est levée, le gestionnaire d'exception analyse les clauses «`catch`» les unes après les autres en commençant par la première. La première clause dont le type correspond à l'exception capture l'exception. Si aucune clause ne correspond, l'exception n'est pas capturée et sera propagée.

En général, les premières clauses **`catch`** correspondent à des exceptions très précises, alors que les dernières sont plus générales.

37.5 LA CLAUSE «FINALLY»

On utilisera la clause «`finally`» (optionnelle) si l'on désire effectuer un traitement dans tous les cas, même si elle est propagée.

```
try {
        On essaye ..
        ..code pouvant lever une exception ..
}
catch (NullPointerException e) {
        Traitement de l'erreur NullPointerException
}
catch (IndexOutOfBoundsException e) {
        Traitement de l'erreur IndexOutOfBoundsException
}
catch (IOException e){
        Traitement de l'erreur IOException
}
finally {
        Ecrire ici le code qui sera exécuté dans tous les cas,
        que l'exception soit ou non levée, que l'exception soit récupérée
        ou propagée.
}
```

Notamment, il est possible d'utiliser la clause «`finally`» pour « remettre de l'ordre ». Par exemple, pour fermer un fichier qui aurait été ouvert dans la clause «`try`».

37.6 Cas particulier : les exceptions de type RunTimeException

Toute méthode susceptible de propager une exception doit le signaler au moyen d'une clause **`throws`** .. En fait, ça n'est pas tout à fait vrai : les exceptions dites « **non contrôlées** » échappent à cette règle.

Appartiennent à cette catégorie les exceptions de type `RunTimeException`. Ainsi, les erreurs de programmation telles que `NullPointerException`, `ArrayOutOfBoundsException`, `ArithmeticException`, etc.. peuvent être « ignorées » par le programmeur. Ignorer de telles exceptions signifie qu'elles seront *propagées*, sans même que le programmeur ait eu la nécessité de le signaler au moyen d'une clause «`throws`».

Ouf ! cette absence de contrôle allège quelque peu l'écriture des programmes. En effet, tout envoi de message est susceptible de lever l'exception «`NullPointerException`» ; imaginons alors un programme qui s'amuse à traiter toutes les exceptions de ce type…

Les exceptions de type «`Error`», - les erreurs système -, sont également non contrôlées.

37.7 Lever une exception

S'opère au moyen de l'opérateur **`throw`**.

Considérons par exemple l'écriture de la méthode setRayon :

```
public void setRayon (int valeur)
                                throws RayonException {
     if (valeur <= 0){
        throw new Exception("le rayon doit être >= 0");
          L'exécution de la méthode s'arrête, l'exception est propagée
          au code appelant
     }
     Valeur du paramètre correcte, suite du code de la méthode
     :
}
```

37.8 CRÉER SON PROPRE TYPE D'EXCEPTION

On peut créer une nouvelle classe d'exception, comme par exemple :

```
class RayonException extends Exception {
     public RayonException() {
          super ("le rayon doit être >= 0") ;
          Invocation du constructeur de la superclasse(classe Exception)
     }
}
```

Utilisation (comparer avec le paragraphe précédent)

```
public void setRayon (int valeur)
                                   throws RayonException {
     if (valeur <= 0){
        throw new RayonException () ;
          L'exécution de la méthode s'arrête, l'exception est propagée au
          code appelant
     }
     Valeur du paramètre correcte, suite du code de la méthode
     :
}
```

Pour quel bénéfice ?

En déclarant la classe «`RayonException`» comme nous l'avons fait, nous avons spécialisé la superclasse «`Exception`» en définissant un message d'erreur spécifique. Autre bénéfice, notre traitement d'erreur gagnera en précision, et nous serons capables notamment grâce à l'instruction «`try-catch`» de différencier l'exception «`RayonException»` de toutes les autres (voir précédemment sous [Différencier les types d'exceptions]).

38 Les paquetages

Java

Comme certaines applications Objet manipulent un nombre de classes gigantesque, le programmeur en arrive très vite à s'arracher les cheveux. Comment mettre de l'ordre? Java s'inspire directement de la conception des systèmes de fichiers : les classes seront organisées en répertoires que l'on appelle des ***paquetages*** (il contiendront chacun un paquet de classes).

Un paquetage Java est un ensemble de *classes* et *d'interfaces* plus ou moins affiliées les unes aux autres, et souvent destinées à coopérer dans le cadre d'une même application ou d'une même fonctionnalité de l'application; ainsi, le paquetage `java.awt` ras-

semble tous les éléments de Java pouvant être affichés dans le cadre d'une interface graphique.

Un avantage du groupage de classes en paquetage est que le risque de **collisions de noms** s'en trouve réduit : chaque paquetage dispose de son propre « espace de noms », permettant ainsi de trouver des classes de même nom dans des paquetages différents.

38.1 Définition et utilisation d'un paquetage

On définit un paquetage en utilisant le mot réservé «`package`», suivi du nom de ce paquetage.

```
package MonPaquetage;
public class ClasseA { ... }
class ClasseB { ... }
```

Dans cet exemple, les deux classes «`ClasseA`», et «`ClasseB`» font partie du paquetage «`MonPackage`».

Le mot-reservé «`package`», s'il figure dans un fichier, doit être le premier mot de la première ligne significative (ni vide, ni commentaire) du fichier source. Si aucun paquetage n'est mentionné, un paquetage anonyme est défini par défaut par le compilateur, nommé «**`unnamed`**». Ainsi, les classes d'un même fichier ne peuvent appartenir qu'à un seul paquetage.

38.2 Utilisation et importation d'un paquetage

Seuls les éléments « exportables » d'un paquetage peuvent être utilisés. En l'occurrence, il s'agit des classes et des interfaces déclarées au sein du paquetage avec le mot-clé **`public`**, ainsi que de ses sous-paquetages.

Pour utiliser une classe publique (ou une interface publique) d'un paquetage, il suffit de préfixer l'identificateur par son nom de paquetage.

Ainsi, on peut utiliser «`ClasseA`» de la manière suivante :

```
class Test {
     void uneMethode() {
          MonPackage.ClasseA mpA = new MonPackage.ClasseA();
          int i = mpA.uneMethodeDeA();
     }
}
```

Un inconvénient évident est qu'il faut retaper le nom du paquetage à chaque fois.

Java permet d'éviter cet excédent de texte au moyen du mot réservé «**import**», qui permet d'ouvrir le contexte d'un paquetage, et d'éviter ainsi la répétition fastidieuse du nom des paquetage utilisés. Ainsi, le petit exemple ci-dessus peut s'écrire :

```
import MonPackage.ClasseA;

class Test {
     void uneMethode() {
          ClasseA mpA = new ClasseA();
          int i = mpA.uneMethodeDeA();
     }
}
```

Il est également possible d'ouvrir le contexte du paquetage entier, et de définir ainsi toutes les classes du paquetage comme importées; la syntaxe est alors la suivante :

```
import MonPackage.*;
import java.awt.*;
```

En cas d'ambiguïté, il est nécessaire d'adresser de manière explicite des classes ayant le même nom dans des paquetage différents.

Il n'est pas nécessaire d'importer le paquetage dont fait partie la classe que l'on utilise: cette importation est automatique. Il n'est pas non plus utile d'importer «`java.lang`» car cette importation est réalisée automatiquement; ce paquetage est tellement fondamental qu'aucune application java ne pourrait s'exécuter sans son concours.

38.3 LES HIÉRARCHIES DE PAQUETAGES

Pour organiser les classes, nous regrouperons ces dernières en paquetages qui eux mêmes pourront contenir d'autres paquetages plus spécialisés et ainsi de suite. Nous réaliserons ainsi une **hiérarchie de paquetages** à la manière des hiérarchies de répertoires dans les systèmes de fichiers.

L'importation d'une class `Xxx` située dans le sous-paquetage `Bbb` du paquetage `Aaa` s'écrira ainsi, en explicitant le chemin d'accès:

```
import Aaa.Bbb.Xxx ;
```

38.4 RELATIONS AVEC LE SYSTÈME DE GESTION DE FICHIERS

Il existe une correspondance rigide entre les dénominations de classes et de paquetages et les noms de fichiers utilisés.

Les règles à respecter..

- Entre la **classe publique** décrite dans **un fichier** et le nom de **ce fichier** : la classe et le fichier portent le même nom, à l'extension «`.java`» près.

- Les paquetages obéissent à une règle similaire. Les fichiers d'un même paquetage **sont placés dans un répertoire unique,** dont le nom correspond à celui du paquetage.

- Ainsi, une **hiérarchie de paquetages** devra en principe mise en place en élaborant une hiérarchie correspondante de répertoires dans le système de fichiers.

---

39 Structure d'une application Java

Comme une application Java peut contenir un nombre relativement énorme de classes, l'outil naturel qui s'offre au programmeur et d'organiser ces classes au sein d'une ou de plusieurs **hiérarchies de paquetages** : nous trouverons dans un même paquetage toutes les classes qui collaborent à une même fonctionnalité.

Chaque paquetage sera placé dans un répertoire du système de fichiers, répertoire qui portera le même nom que le paquetage lui-même.

39.1 Composition d'un paquetage

Les classes et les interfaces qui composeront le paquetage seront écrites dans des **fichiers** portant l'extension «`.java`». Ces fichiers seront tous placés dans le même répertoire, celui du paquetage.

39.2 Composition d'un fichier

En général, chaque fichier contiendra **une seule classe publique** (ou une seule interface publique). Si le mot `public` n'est pas mentionné, cette classe (ou interface) ne sera visible qu'à l'intérieur du paquetage. Tout élément exportable doit être désigné par le mot-clé `public`.

Un fichier peut contenir un nombre indéterminé de classes et d'interfaces !

Le rôle du fichier est de mettre à disposition **une** classe ou **une** interface qui sera utilisée par les autres composants de l'application.

Si la mise en oeuvre de cette classe (ou interface) nécessite l'écriture d'autres classes ou interfaces, écrits dans le seul but de prendre en charge une partie du travail, le programmeur les placera naturellement dans le même fichier.

Un seul composant `public` par fichier !

Les « composants secondaires », nécessaires à la mise en oeuvre du composant principal public, ne seront jamais visibles en dehors du paquetage. En effet, Java impose que chaque fichier ne contienne qu'un seul élément exportable.

Rappelons que le nom du fichier devra correspondre exactement au nom du composant principal, accompagné de l'extension «`.java`».

39.3 AMORCE DE L'APPLICATION: LA CLASSE PRINCIPALE (`main`)

Parmi les classes publiques, une au moins[1] comportera une fonction «`main`» qui sera invoquée au moment du lancement du programme :

```
> java Xxx
```

Ceci provoque l'exécution de la machine virtuelle Java, qui invoque la fonction «`main`» de la classe «`Xxx`» (qui doit être une classe publique).

39.4 FONCTION `main`, PARAMÈTRES EN LIGNE DE COMMANDE

La fonction `main` doit avoir obligatoirement l'allure suivante :

```
public static void main (String[] args) {
     // corps de la fonctions main
     :
}
```

Les paramètres passés à la fonction «`main`» sont les paramètres spécifiés par l'utilisateur du programme dans la ligne de commande lorsque le programme est lancé.

Un programme peut par exemple lancé ainsi :

```
RepertoireCourant> java Xxx 10 Bonjour Alfred
```

Le noyau Java transmet les paramètres à la fonction «main» sous forme de **chaînes de caractères**. Ainsi, le paramètre «**`args`**» de la fonction «main» est un tableau de chaînes de caractères (`String`). Dans notre exemple, la première case de ce tableau («`args[0]`») contiendrait le texte «10».

1. Plusieurs classes peuvent comporter une fonction `main` (ce qui permet de les tester indépendamment les unes des autres).

Voici une fonction «main» qui affiche le ou les paramètres passés par l'utilisateur dans la ligne de commande:

```
public static void main (String[] args) {
     if (args.length >= 1) {
          System.out.println
          ('\n' + "Appel avec " + args.length + c parametres:");
          for (int cpt = 0; cpt < args.length; cpt++) {
               System.out.println (cpt + "> " + args[cpt]);
          }
     }
}
```

Notons pour finir que chaque classe Java peut être dotée d'une fonction «main», ce qui l'autorise à fonctionner en tant qu'application « standalone». Ceci est très pratique pour tester toutes les méthodes de la classe dans la phase de mise au point.

40 Localisation des classes et compilation

Voyons comment rendre nos classes accessibles aux programmes qui les utiliseront..

Pour simplifier le langage, nous utiliserons le terme **librairie** pour dénoter une hiérarchie de paquetages, comprenant un paquetage principal, lui-même composé de sous-paquetages etc..

Supposons que nos classes soient placées dans un ou plusieurs répertoires spéciaux, correspondant chacun à une **librairie**. Comme par exemple :

- /home/projetX/mesclasses1
- /home/projetX/mesclasses2
- /home/lib

Notons que chacun de ces 3 répertoires est un **répertoire de base** pour chaque arborescence de paquetages.

40.1 COMPILATION DU PROGRAMME

Pour compiler le programme, il nous faut maintenant définir le « classpath », c'est-à-dire le chemin des classes, qui est la collection de tous les *répertoires de base*.

Si on utilise l'environnement Java SDK pour compiler, deux solutions s'offrent à nous pour définir le chemin des classes :

1/ Définir la variable d'environnement CLASSPATH au niveau du système d'exploitation ;

2/ Ou (solution la plus souple et la plus courante), utiliser l'option –classpath du compilateur javac.

Par exemple, sous Windows, la deuxième solution s'écrirait ainsi, **sur une seule ligne** :

```
>javac –classpath .;c:\home\projetX\mesclasses1;
c:\home\projetX\mesclasses1;c:\home\lib MonPro-
gramme.java
```

Sous Linux, le point-virgule est remplacé par «**:**» et les «****» deviennent des «**/**», ce qui donnerait :

```
>javac –classpath .:/home/projetX/mesclasses1:/home/
projetX/mesclasses1:/home/lib MonProgramme.java
```

Dans les deux cas, le point «**.**» désigne le **répertoire courant**, là où se trouve généralement le programme à compiler (`MonProgramme.java`). Par défaut, si on ne précise pas l'option -classpath, le compilateur javac cherche les classes dans le répertoire courant.

La recherche des classes s'opère de répertoire de base en répertoire de base, dans l'ordre défini par l'option –classpath.

☺ La compilation est partielle : le compilateur ne compile que les classes qui méritent une compilation ! A cette fin, il consulte tout d'abord le fichier source pour voir si la source est plus récente que le fichier classe. Si oui, une compilation est opérée.

Et si les sources ne sont pas situées au même endroit que les classes ?

Il suffit pour cela d'utiliser l'option –sourcepath du compilateur, en indiquant le chemin des sources (de manière identique au chemin des classes) . Considérons par exemple le cas où le répertoire courant contient deux sous-répertoires : src pour les sources et classes pour les classes. La compilation s'écrira sur une seule ligne:

```
>javac –classpath .\classes;cheminDesLibrairies –sour-
cepath .\src MonProgramme.java
```

40.2 EXÉCUTION DU PROGRAMME PAR LA MACHINE VIRTUELLE

Le principe est le même que pour la compilation. Ainsi, en optant pour la deuxième solution, on écrira sous Windows :

```
>java –classpath .;c:\home\projetX\mesclasses1;
c:\home\projetX\mesclasses1; c:\home\lib MonProgramme
```

Il est possible toutefois d'écrire –**cp** en lieu et place de `-classpath`.

40.3 LES FICHIERS D'ARCHIVES JAR

☺ Une archive JAR contient plusieurs classes, plusieurs paquetages hiérarchisés ou non dans un seul fichier. Les données y sont compressées au format ZIP. Ainsi, l'utilisation d'une archive JAR est une économie d'espace et de temps d'accès (un seul accès fichier).

Par exemple, les milliers de classes de la bibliothèque d'exécution du JDK se trouvent rassemblées dans l'archive rt.jar, qui se trouve placée dans le répertoire jre/lib de l'environnement Java SDK.

En tant que développeur, on peut utiliser l'utilitaire jar pour créer nos propres archives. Ce dernier est fourni avec l'environnement Java SDK. Voir à ce propos l'annexe au point 40.3, page 51.

Par exemple, supposons que l'on ait créé l'archive `arch.jar`, placée dans le répertoire de base c:\home\archives, à partir des deux librairies classes1 et classes2 de l'exemple précédent. Il faudra mettre à jour le chemin de classes aussi bien pour la compilation que pour l'exécution, comme par exemple :

```
>javac –classpath .;c:\home\archi-
ves\arch.jar;c:\home\lib MonProgramme.java
```

☺ Par défaut, les classes sont toujours recherchées dans les fichiers de bibliothèque d'exécution (rt.jar) et les autres JAR dans les répertoires jre/lib et jre/lib/ext, que ce soit pour la compilation comme pour l'exécution.

40.4 STRUCTURE D'UNE CLASSE

Java La déclaration d'une classe obéit à la syntaxe suivante :

```
[Modificateurs de classe] class NomDeLaClasse
                [extends nomDeLaSuperclasse]
                 [implements  interface1,  interface2,..]
{

     [Variables]
     [Constructeurs]
```

```
            [Méthodes]
        }
```

☺ Notons que les variables, constructeurs et méthodes peuvent être déclarés dans n'importe quel ordre. Par convention, on préfère commencer par déclarer les variables, puis les constructeurs pour finir par les méthodes.

Un exemple ..

```
public class CreatureMythique
                  extends Creature
                  implements Runnable {
// Variables de classe (voir plus loin, le point [Variables de classe] )
private static int dureeSieste ;
// Variables d'instance
     private int age ;       // Age de la créature
     private String nom ;    // Nom de la créature
     private estMagique ;    // Existence de pouvoirs magiques
     private Thread ;        // Activité propre (objet actif)


// Constructeurs
     public static {
     // Constructeur de classe, pour initialiser les variables de classe
          dureeSieste = 1000;
     }

     public CreatureMythique (    String nom,
                                  int age,
                                  boolean estmMagique) {
     // Constructeur d'instance, pour initialiser les variables d'instance
          this.age = age ;
          this.nom = nom ;
          this.estMagique = estMagique;
          activite = new Thread(this) ;
          activite.start();
     }

// Méthodes

     public static void setDureeSieste (int duree) {
     // Une méthode de classe
          dureeSieste = duree;
     }

     public void setAge (int age) {
     // Une méthode d'instance
          this.age=age;
     }
```

```
    public void run() {
    // Implémentation de l'interface Runnable
        while(true) {
            System.out.println ("Bouh !! ");
            try {Thread.sleep(dureeSieste);}
            catch (InterruptedException e) {}
        }
    }

}
```

40.5 LES MODIFICATEURS DE CLASSE

Les modificateurs de classe sont des mot-clés optionnels qui précèdent le mot-clé «`class`». Ces modificateurs spécifient la portée ainsi que le statut de la classe au sein du programme.

Le langage prévoit 3 modificateurs de classe : «`abstract`» «`final`» et «`public`» .

- Le modificateur « **`public »`** annonce que cette classe peut être utilisée - notamment instanciée ou héritée - partout dans le même paquetage («`package`») ou alors dans tout autre paquetage qui «importerait» cette classe. Pour plus de détails, le lecteur est renvoyé au point 38, page 44.

  Toute classe publique doit être déclarée dans son proche fichier qui doit porter obligatoirement le nom de la classe publique : <`nomDeLaClasse`>`.java`. Un fichier ne peut contenir qu'une seule classe publique. Un paquetage peut être composé de plusieurs fichiers.

  Si le mot-clé «`public`» n'est pas utilisé, le mode «`package`» est sous-entendu : la classe est considérée simplement comme une «classe amie», qui ne peut être utilisée qu'à l'intérieur du même paquetage, mais qui ne peut être utilisée ailleurs, même par le biais d'une importation.

- Le modificateur « **`abstract »`** annonce qu'il s'agit d'une «classe abstraite», qui définit un comportement commun à une hiérarchie de classes dérivées par héritage. *Le concept d'héritage n'est pas décrit dans le cadre de ce résumé.*

- Le modificateur «**`final`**» annonce que la classe ne peut pas avoir de «sous-classes»: aucune classe peut en hériter à des fins de spécialisation.

40.6 «EXTENDS» : HÉRITAGE

Le concept d'héritage n'est pas décrit dans le cadre de ce résumé.

Ce mot-clé est optionnel.

Pour éclairer le lecteur, disons simplement que le mot-clé «`extends`» indique que la classe hérite de la classe «`Creature`». Grâce à cet héritage, toutes les opérations applicables aux objets de la classe «`Creature`» s'appliquent automatiquement aux objets de la classe «`CreatureMythique`».

☺ Si le programmeur omet le mot-clé «`extends`» lors de la déclaration d'une classe, le compilateur Java assume que la nouvelle classe hérite directement de la classe «`Object`». Tout en haut de la hiérarchie, on trouve donc la superclasse «`Objet`» dont hérite toutes les classe Java.

40.7 «IMPLEMENTS» : IMPLÉMENTATION D'INTERFACE

Le concept d'inetrface n'est pas décrit dans le cadre de ce résumé.

Ce mot-clé est optionnel.

Pour éclairer le lecteur, disons simplement qu'une peut «implémenter» une «interface», voire même plusieurs... Une interface est une **liste de méthodes** décrites uniquement par leur entête.

Dans l'exemple présenté plus haut, il est fait mention de l'interface `Runnable`, une interface de la librairie Java, spécifiée comme suit :

```
interface Runnable {
     public void run() ;
}
```

Une classe qui implémente une interface s'engage à fournir le code de toutes les méthodes décrites dans l'interface. C'est ainsi que nous retrouvons la méthode `run()` avec ses instructions au sein même de la classe `CreatureMythique.`

En l'occurrence, les objets de type «`Runnable`» sont appelés des ***objets actifs***, caractérisés par un flot d'exécution concurrent au reste du programme. La méthode «`run`» qui les caractérise contient le code de l'activité concurrente.

41 Visibilité des entités au sein d'une classe

Commençons par un exemple illustrant la visibilité des variables (le concept est généralisable aux méthodes):

```
public class UneClasse {

     private int v1 ; // une variable d'instance privé, visible:
                      . partout à l'intérieur de la classe
                      . et c'est tout..

     protected int v2 ; // une variable d'instance protégée, visible :
                      . partout à l'intérieur de la classe
                      . dans les classes extérieures de la même hiérarchie
                        d'héritage
                      . dans les classes extérieures du même paquetage

     public int v3 ; // une variable d'instance publique, visible :
                      . partout à l'intérieur de la classe
                      . dans toutes les classes extérieures

     int v4 ;  // mode « package », par défaut ..
               une variable d'instance protégée, visible :
               . partout à l'intérieur de la classe
               . dans toutes les classes extérieures du même paquetage


     public void uneMethode(int p) {
     //    Le paramètre formel « p » est local à la méthode : il n'est visible
           qu'à l'intérieur de la méthode
           int v ;   //    Variable locale à la méthode : visible uniquement
                           à l'intérieur de la méthode, à partir de l'endroit /où cette
                           variable est déclarée
           .. suite de la méthode ..
     }
}
```

Dans ce qui suit, le terme « **élément** » dénote une *variable* (variable d'instance ou variable de classe) ou une *méthode* (méthode d'instance ou méthode de classe).

- **Concernant les éléments définis au sein même de la classe**

  En **ada**, la visibilité d'un identificateur commence à partir de l'endroit où se trouve déclaré l'identificateur en question.

  ☺ En **java**, la visibilité est globale à toute la classe : les variables et les méthodes peuvent être utilisées avant d'avoir été déclarées !

- **Concernant les éléments définis par une classe extérieure**

  Leur accès est contrôlé par leur mode d'accès `public`, `private`, `protected` ou encore « `package` » (le mode d'accès « par défaut », quand rien n'est précisé).

  Sont accessibles les éléments dont le mode d'accès est **`public`**, ou encore **`protected`** si la classe en question est une superclasse (héritage..).

  Si la classe extérieure appartient au même paquetage, sont accessibles également les éléments dont le mode d'accès est « **`package`** ».

  Etre attentif au fait que les éléments d'une classe extérieure ne sont visibles que si la dite classe est elle-même visible ! Une classe extérieure est visible:

  - si elle appartient au **même paquetage** ;
  - ou alors s'il s'agit d'une classe publique **importée** (depuis un autre paquetage, voir le point 38, page 44).

- **Concernant les paramètres formels et les variables locales d'une méthodes**

  Ces derniers ne sont visibles qu'au sein même de la méthode, invisibles à l'extérieur de la méthode (concept identique à ada).

42 Les constructeurs

Java

Un constructeur est une méthode un peu spéciale utilisée pour initialiser les objets lors de leur création. Un constructeur est invoqué automatiquement lorsque l'opérateur «**new**» est exécuté.

Voici par exemple la classe «`CreatureMythique`», pourvue de trois constructeurs :

```
class CreatureMythique extends Creature {
// Variables d'instance
      private int age ;      // Age de la créature
      private String nom ;   // Nom de la créature
      private estMagique ;   // Existence de pouvoirs magiques

// Constructeurs
     public CreatureMythique () {
     // Premier constructeur: un constructeur sans paramètre
          nom = "Anonyme";
     }

     public CreatureMythique (    String nom,
```

```
                                    int age,
                                    boolean estmMagique) {
    // Un deuxième constructeur, avec paramètres
         this.age = age ;
         this.nom = nom ;
         this.estMagique = estMagique;
    }
    public CreatureMythique (CreatureMythique cr) {
    // Un troisième constructeur : un constructeur « de copie »
         this(cr.age, cr.monNom , cr.estMagique);
         //    Le mot-clé « this » permet d'invoquer 'un
               autre constructeur de la même classe (en l'occurrence,
               le deuxième constructeur)
    }

    :
    :
}
```

Attention aux points suivants :

- Le nom du constructeur doit être rigoureusement identique à celui de la classe elle-même.
- Un constructeur ne retourne aucune valeur. Un constructeur se comporte comme une procédure : le mot-clé «`void`» est sous-entendu.
- Un constructeur n'admet que l'un ou l'autre des 3 modificateurs d'accès «`public`», «`protected`» ou «`private`». Les modificateurs «`abstract`», «`final`» ou «`static`» ne sont pas utilisables.

42.1 RÔLE DÉTAILLÉ D'UN CONSTRUCTEUR

Voir le point 20, page 16

42.2 ENCHAÎNEMENT DES CONSTRUCTEURS

Le constructeur est une méthode qui sert principalement à initialiser les variables d'instance. Qu'en est-il des variables d'instance héritées : comment et avec quelles valeurs sont-elles initialisées ?

Lors de la création d'un objet, l'invocation des constructeurs s'enchaîne en commençant par le constructeur de plus haut niveau : celui de la classe «Object». Ceci entraîne l'initialisation de toutes les parties héritées de la classe.

Pour ce faire, un constructeur commence toujours par invoquer le constructeur de la superclasse, et ainsi de suite..

 Cette invocation doit constituer la toute première instruction d'un constructeur !

Comment invoquer le constructeur de la superclasse ?

En Java, le mot-clé «**super**» permet d'invoquer le constructeur de la superclasse.

Pour une classe nommée «Xxx», le programmeur écrira un ou plusieurs constructeurs qui auront typiquement l'allure suivante :

```
public Xxx (paramètres) {
     super(paramètres) ;        // Invocation du constructeur
                                // de la superclasse
     .. initialisation des variables de la classe Xxx ..
}
```

Insistons sur le fait que l'invocation du constructeur de la superclasse doit obligatoirement constituer la toute première instruction !

☺ Invocation implicite du constructeur de la superclasse

Le programmeur peut omettre l'invocation au constructeur de la superclasse. Cette invocation **sera opérée automatiquement** par Java au moyen de l'instruction :

```
super() ;
```

Attention toutefois.. Cette omission est possible uniquement si la superclasse possède un « constructeur sans paramètre » (ce qui permet au noyau Java de l'invoquer sans avoir à réfléchir ..)

Dans le cas contraire, le programmeur est tenu par le compilateur d'opérer une invocation explicite lui permettant de préciser la valeur des paramètres. Le compilateur le vérifie.

42.3 LE CONSTRUCTEUR PAR DÉFAUT

Si le programmeur a écrit une classe sans y avoir spécifié aucun constructeur, le compilateur génère automatiquement un ***constructeur par défaut*** , sans paramètre, qui ne fait rien d'autre que d'invoquer le constructeur sans paramètre de la superclasse.

Pour une classe nommée «Xxx», ce constructeur par défaut aurait l'allure suivante:

```
public Xxx () {
        super() ; //   Invocation du constructeur sans paramètre
                      de la superclasse
}
```

On peut se rendre compte que la génération automatique d'un constructeur par défaut n'est possible que si la superclasse dispose d'un constructeur sans paramètre.

Dans le cas contraire, le programmeur est invité par le compilateur à corriger son erreur: il doit déclarer explicitement **au moins** un constructeur avec ou sans paramètre.

Notons enfin qu'à partir du moment où le programmeur a défini un constructeur avec ou sans paramètre, le compilateur ne génère aucun constructeur par défaut. C'est la raison pour laquelle une classe ne possède pas forcément de constructeur sans paramètre.

42.4 LE CONSTRUCTEUR DE CLASSE

L'équivalent du *constructeur* existe pour les variables de classe.

Un tel constructeur est désigné par le mot-clé «**static**». Ce constructeur sera invoqué au moment du chargement de la classe.

Par exemple :

```
static {
     :
     v = 20 ;  // Initialisation d'une variable de classe
     :
}
```

A l'instar des *méthodes de classe*, le constructeur de classe ne peut accéder qu'aux variables de classes. L'accès aux variables d'instance (informations propres aux objets) lui est interdit.

43 Les destructeurs

Java

Les destructeur est optionnel. En Java, l'utilisation d'un destructeur est assez rare. En effet, le rôle accompli par le *garbage collector* est suffisant dans la plupart des cas.

Si le programmeur désire effectuer certaines actions (des libérations ou autres..) au moment ou l'objet est détruit par la machine Java, il a la possibilité de redéfinir la méthode `finalize` qui devra être déclarée ainsi :

```
public void finalize () {
          .. mes dernières volontés ..
}
```

Cette méthode est appelée par le garbage collector juste avant que l'objet ne soit détruit.

Note C ++

En C++, le destructeur est très important (pas de garbage collector !). Ce dernier porte le même nom que le constructeur à l'exception près d'un « ~ » placé juste devant le nom.

44 Variables de classes et variables d'instance

Java

Les **variables de classe** sont utilisées pour mémoriser des **informations globales** qui concernent la classe elle-même et qui sont partagées par toutes les instances.

Une variable de classe est introduite par le mot-clé «**`static`**».

En mémoire, une variable de classe n'existe **qu'en un seul exemplaire**, quel que soit le nombre objets instanciés.

Les **variables d'instances** sont, - comme leur nom l'indique -, propres à chaque instance : un jeux de variables d'instance existera en mémoire pour chaque instance créée.

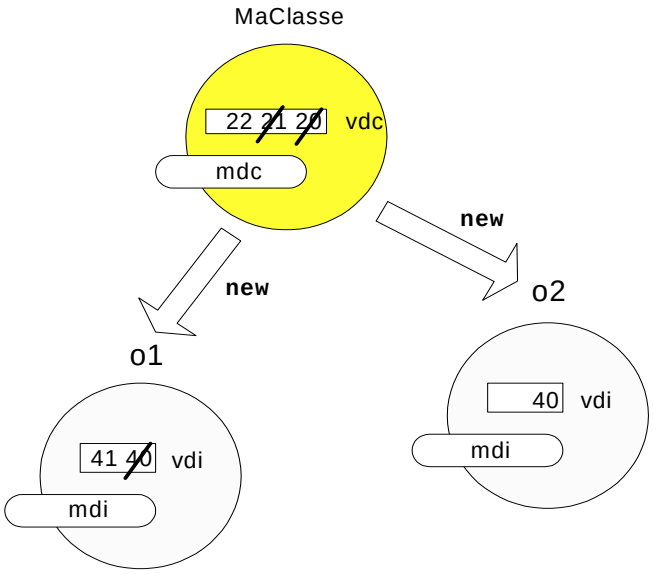
```
class MaClasse {

    private static int vdc ;// une variable de classe
```

```
        private int vdi ;  // une variable d'instance

        public static {
        // Constructeur de classe : pour initialiser les variables de classe
             vdc = 20 ;
        }

        public MaClasse() {
        //Constructeur classique : pour initialiser les variables d'instance
             vdi = 40 ;
        }

        public static void mdc () {
        // une méthode de classe
             vdc++;
             vdi++;    //  une méthode de classe ne peut pas accéder aux variables d'instances
        }
        public void mdi() {
        // une méthode d'instance classique
             vdc++;
             vdi++;
        }
}
```

Imaginons les quelques lignes de code suivantes:

```
MaClasse o1 = new MaClasse() ;
MaClasse o2 = new MaClasse() ;
MaClasse.mdc();     // Invocation de la méthode de classe
o1.mdi() ;          // Invocation de la méthode d'instance
```

En mémoire, nous obtenons le schéma suivant :



Comme deuxième exemple, voici une classe qui compte ses instances : à chaque création d'objet, le constructeur, qui est invoqué automatiquement, incrémente la variable d'instance «nombreDInstances».

```
public class MesInstancesSontComptees {
     // Variable de classe comptant le nombre d'instances
     private static int nombreDInstances = 0;

     MesInstancesSontComptees () {
     // Constructeur
     // Incrémente le compteur à chaque création d'instance
          nombreDInstances++ ;
     }

     public static int nbInstances () {
     // Méthode d'accès à la variable de classe
          return nombreDInstances;
     }
}
```

44.1 INITIALISATION D'UNE VARIABLE DE CLASSE

De manière identique aux variables d'instance, la déclaration d'une variable de classe peut comporter un «initialiseur» :

```
static int v = 10 ;
```

Cette initialisation aura lieu au moment du «chargement de la classe», c'est à dire dès l'instant que la classe est utilisée.

44.2 ACCÉDER AUX VARIABLES DE CLASSE ET AUX VARIABLES D'INSTANCES

Les **variables d'instance** ne peuvent être accédées que par les **méthodes d'instance**. En effet, comme le montre la figure du premier exemple (voir plus haut), ce sont les méthodes d'instance qui peuvent agir directement sur les instances.

Par contre, une méthode de classe ne peut agir qu'au niveau de la classe : elle n'a pas accès aux variables d'instance qui existent en plusieurs exemplaires, avec un exemplaire par instance.

☺ Notons que les **variables de classe** sont accessibles aussi bien par les méthodes de classe que par les méthodes d'instance. Une variable de classe est ainsi partagée par toutes les instances.

44.3 LE CONSTRUCTEUR DE CLASSE

L'équivalent du «constructeur» existe pour les variables de classe.

Un tel constructeur est désigné par le mot-clé «**static**». Ce constructeur sera invoqué au moment du chargement de la classe. Comme ci-après :

```
static {
     :
     v = 20 ;  // Initialisation d'une variable de classe
     :
}
```

A l'instar des *méthodes de classe*, le constructeur de classe ne peut accéder qu'aux variables de classe. L'accès aux variables d'instance (informations propres aux objets) lui est interdit.

45 Déclarer des constantes symboliques : `final`

Les constantes symboliques (comme PI = 3.14) sont déclarées comme des **variables** (eh oui !) qui comportent **une valeur d'initialisation non modifiable.**

Il suffit donc de déclarer une variable en précisant qu'elle est non modifiable au moyen du mot-clé final.

```
class MaClasse {
          ..
          final double PI = 3.14 ;
          ..
}
```

☺ Par convention d'écriture, les constants symboliques sont écrites en majuscules

☺ Pour économiser la mémoire vive, les constantes symboliques sont le plus souvent déclarées en tant que **variables de classe** afin de ne pas être dupliquées inutilement au sein de chacune des instances de la classe (voir le point 44, page 60):

```
class MaClasse {
          ..
          static final double PI = 3.14 ;
          ..
}
```

46 Partage d'objets, copie superficielle et copie profonde

Ces différents concepts sont propres à la programmation Objet et ils sont tout particulièrement importants en Java, un langage qui manipule les objets de manière dynamique.

46.1 Retour sur l'affectation

1/ avec les primitifs : la copie de valeur

```
int x, y;
x = 3;
y = x;
y = 5;
```

x 0 y 0

x 3 y 0

x 3 y 3

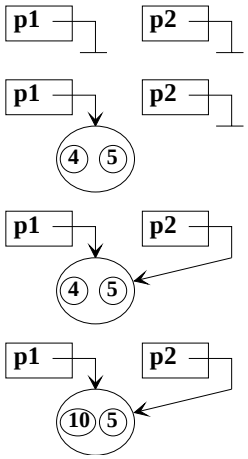
x 3 y 5

L'affectation de types primitifs opère une **copie de valeur** ;

Par ailleurs, les valeurs des deux variables restent indépendantes l'une de l'autre : la modification de `y` ne concerne en rien la valeur de `x`.

2/ avec les objets : le partage d'objet

```
Point p1, p2;
p1 = new Point (4, 5);
p2 = p1;
p2.x = 10;
```



L'affectation d'objets opère un **partage d'objets** : le même objet est référencé par deux variables différentes.

Par la suite, la valeur de l'objet peut être modifiée en passant par l'une ou l'autre des deux variables

46.2 COPIE D'OBJETS

En Java, la copie d'objets nécessite l'écriture de méthodes spécifiques. En effet, le concept n'est pas aussi immédiat qu'il n'y paraît de prime abord et le programmeur devra déterminer lui-même le type de copie qu'il désire mettre à disposition:

- une **copie superficielle** (« **shallow copy** »),
- ou une **copie profonde** (« **deep copy** ») qui opère de manière récursive en effectuant également une copie profonde des sous-objets,
- ou encore un intermédiaire entre ces deux types ..

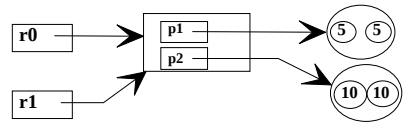
Exemple : les classes «`MyPoint`» et «`MyRectangle`»

```
class MyPoint {
  public int x, y;

  public MyPoint(int x, int y) {
      this.x = x; this.y = y;
  }

  public MyPoint deepCopy() {
      return new MyPoint(this.x, this.y);
  }
}
```

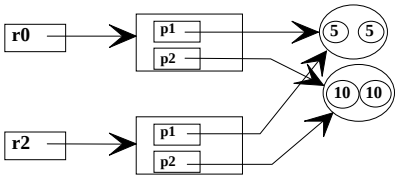
```
class MyRectangle {
  public MyPoint p1, p2;

  public MyRectangle (MyPoint p1, MyPoint p2) {
      this.p1 = p1; this.p2 = p2;
  }

  public MyRectangle shallowCopy() {
   // Copie superficielle
    return new MyRectangle(this.p1, this.p2);
  }

  public MyRectangle deepCopy() {
  // Copie profonde
    return new MyRectangle(this.p1.deepCopy(),
                           this.p2.deepCopy());
  }
}
```

```
public class ObjectCopy {
     public static void main (String [] arg) {
          MyPoint  p1 = new  MyPoint(5, 5);
          MyPoint  p2 = new  MyPoint (10, 10);

          MyRectangle  r0, r1, r2, r3;

          r0 = new  MyRectangle (p1, p2);
```

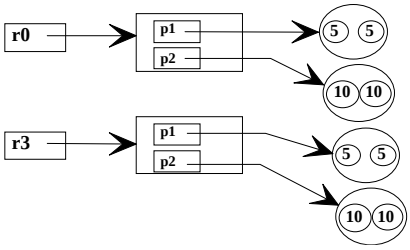
```
          r1 = r0;

          // Partage d'objets
```




```
          r2 = r0.shallowCopy  ();

          // Copie superficielle
          //(limitée à 1 niveau)
```




```
          r3 = r0. deepCopy  ();

         // Copie profonde (récursive)
  }
}
```



46.3 LA MÉTHODE «CLONE» : COPIE SUPERFICIELLE

Au cas où la copie superficielle convient aux besoins du programmeur, la classe `Object` met à disposition la méthode suivante :

```
protected Object clone() throws CloneNotSupportedException;
```

La méthode **`clone()`** permet d'opérer des copies d'objets **bit à bit**, et s'utiliserait par exemple ainsi :

```
objet = (Point)unPoint.clone();
```

Toutefois, les choses ne sont pas si simples, et l'instruction écrite ci-dessus **sera refusée par le compilateur** !! En effet, la méthode clone proposée par la classe `Object` à un mode d'accès protégé (`protected`) : seule la classe `Point` peut cloner des objets de type `Point` !

Pourquoi compliquer les choses ?

Comme la duplication d'objet est une chose délicate (copie superficielle ? copie profonde ? ..), Java invite le programmeur à définir **ses propres méthodes de clonage, dûment réfléchies...**

Le principe préconisé par le langage consiste à obliger le programmeur à redéfinir la méthode `clone` dans la classe même de l'objet et en lui donnant un mode d'accès public.

Voici un exemple typique proposant une copie superficielle simple:

```
class MyPoint implements Cloneable {
:

  public Object clone() {
         try {return super.clone(); }
         catch (CloneNotSupportedException e)
              {return null;}
  }

}
```

La classe `MyPoint` doit obligatoirement **implémenter l'interface `Cloneable`**. En effet, la première chose que fait la méthode `clone` de la classe `Object` est de tester si le type de l'objet courant pour laquelle elle est appelée implémente `Cloneable`. Dans le cas contraire, l'exception `CloneNotSupportedException` est levée.

Notons au passage que l'interface **`Cloneable`** est vide : elle n'a pas de méthode !

Il s'agit en effet d'une « interface de balisage » dont le seul objectif est de permettre l'utilisation de l'opérateur `instanceof` dans une requête comme :

```
if (unObjet instanceof Cloneable) ..
```

47 Annexe A - Les Applets

Une applet est une petite application que l'on exécute au travers du navigateur WEB : Netscape, Explorer ou autre. Comme l'application est référencée par une page HTML, son exécution sera lancée par le navigateur lui-même, au moment du chargement de la page .

47.1 UN BREF APERÇU

Pour le programmeur, réaliser une applet n'est pas plus compliqué que de réaliser une application autonome. Voici par exemple le code d'une applet dont l'exécution affichera le texte « Hello » :

```
import java.applet.Applet;
import java.awt.Graphics;

public class Applet1 extends Applet {

  public void paint(Graphics g) {
      //Dessine un rectangle autour de la zone de texte
      g.drawRect(0, 0,      size().width - 50,
                            size().height - 50);

      //Affiche le texte à l'intérieur du rectangle
      g.drawString("Hello", 100, 100);
  }
}
```

Pour exécuter cette applet, la page HTML interprétée par le navigateur aura l'allure suivante:

```
<HTML>
     <HEAD>
          <TITLE> Hello </TITLE>
     </HEAD>
     <BODY>
          <APPLET   CODE = Applet1.class
                    WIDTH = 600
                    HEIGHT=400>
          </APPLET>
     </BODY>
</HTML>
```

Nous trouvons la balise <**APPLET**> dans le corps même du fichier HTML. Cette dernière sera utilisée pour référencer notre applet. Les paramètres indiquent que l'applet est insérée dans une zone de la page HTML qui aura une largeur de 600 pixels, et une hauteur de 400 pixels.

Pour exécuter cette applet, nous avons deux possibilités :

1/ Charger la page HTML correspondante au moyen du navigateur WEB. L'applet sera alors automatiquement exécutée ;

2/ Utiliser l'utilitaire «`appletViewer`» offert avec le JDK (« Java Development Kit »), mis à disposition pour tester très simplement les applets en dé-

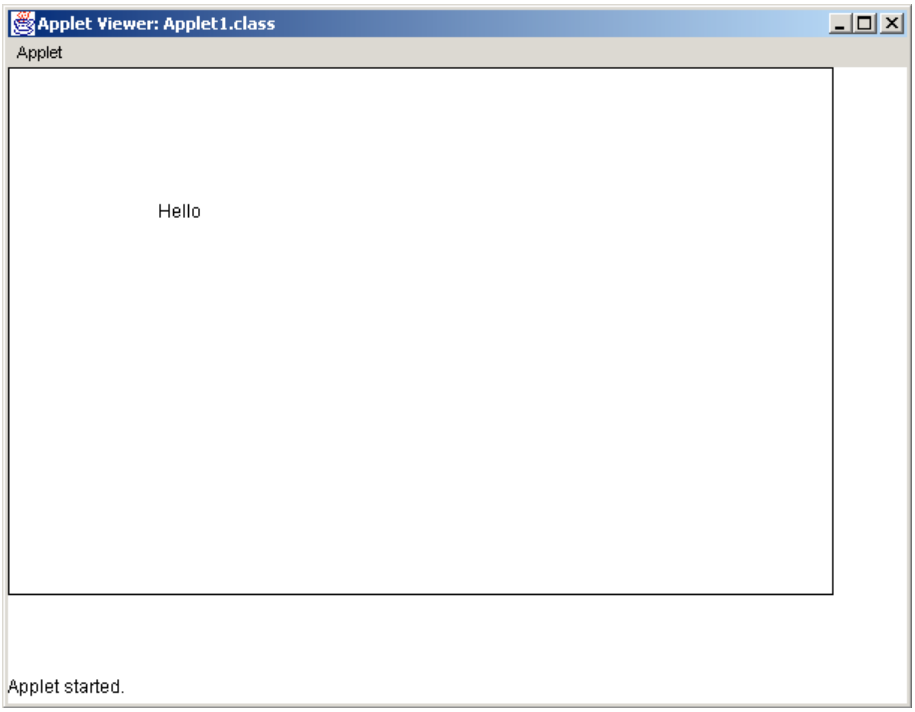
veloppement.

Il suffira de taper la commande :

```
>appletViewer hello.html
```

où «`hello.html`» désigne le fichier HTML écrit ci-dessus.

Voici par exemple son exécution au moyen de l'utilitaire «`appletViewer` »:



Intérêt des applets

Leur principal intérêt est d'étendre encore l'attractivité du réseau : ne plus se limiter à des **échanges d'information**, mais offrir la possibilité **d'échanger des applications**, téléchargées à partir d'un serveur distant pour être exécutées localement, sur le poste client.

Comme le serveur ignore tout de la machine client, - il peut s'agir d'une machine UNIX, d'une machine Windows ou Windows NT -, il faut alors que le navigateur puisse s'appuyer sur une machine virtuelle Java, installée localement, et qui sera capable d'exécuter l'applet en interprétant chacune de ses instructions.

La sécurité

Un problème crucial est alors posé : le téléchargement de l'application sur le poste client doit se faire en toute sécurité. Notamment, le poste client doit être protégé contre

l'exécution de toute application qui violerait l'intégrité de son environnement.

Pour cette raison, le navigateur imposera à l'applet certaines conditions de fonctionnement. Ainsi, du côté client, le contrôle s'effectuera tant au niveau de la mémoire vive, que du disque local, que vers le réseau. Voir pour plus de détails le point 47.5, page 79.

Les applets ne sont plus à la mode

Le temps de téléchargement des applets et la complexité du système de sécurité sont malheureusement extrêmement ennuyeux, et l'on préfère aujourd'hui utiliser des technologies basées sur la création dynamique de pages html au moyen de servlets ou de JSP.

Toutefois, si l'on se cantonne au monde intranet, - caractérisé par des débits élevés et pour lequel les problèmes de sécurité se posent de manière différente -, les applets gardent tout leur intérêt et offrent des possibilités nettement supérieures aux autres technologies.

47.2 LE CYCLE DE VIE D'UNE APPLET

A la différence des applications autonomes, on peut constater que les applets ne possèdent pas de fonction «**`main`**». Comme nous le verrons par la suite, ce point d'entrée sera remplacé dans les applets par deux procédures: **`init`** et **`start`**.

Une applet peut se trouver dans différents états en fonction de l'état de la page HTML qui la référence.

Remarquons en préliminaire qu'un navigateur Internet manipule les pages HTML au moyen d'une pile : quand une nouvelle page est chargée, cette dernière est placée au sommet de la pile ; puis est interprétée par le navigateur.

Par la suite, l'utilisateur peut opérer un parcours de la pile en avant ou en arrière, au grès de ses désirs.

Et ainsi, nous pouvons donc définir deux états pour une page HTML qui aura été chargée par le navigateur :

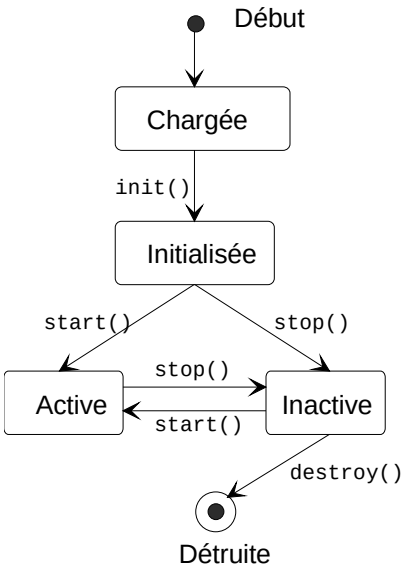
1/ **page active** : en cours d'interprétation ;

2/ **page inactive** : dans le cas contraire.

Nous retrouverons les deux états **Active** et **Inactive** dans le cycle de vie d'une applet. Une applet connaît par ailleurs 2 états supplémentaires : **Chargée**, et **Initialisée.**

Voici le diagramme d'état correspondant au cycle de vie d'une applet :

Figure 1: Cycle de vie d'une applet



Début	Début du chargement de l'applet (« loading »)
Chargée	L'applet a été chargée en mémoire, l'objet représentant l'applet a été instancié en utilisant le constructeur sans paramètre (constructeur par défaut, ou constructeur spécifié par le programmeur), tout l'environnement de l'applet est en place : environnement graphique, possibilité de travailler avec le réseau, ..
Initialisée	Dès que l'applet est chargée et par conséquent que tout son environnement est en place, la machine virtuelle lui envoie aussitôt le message «`init()`» (voir ci-après). Après exécution de la méthode «`init`», l'applet passe à l'état Initialisée.
Active, Inactive	Ces deux états reflètent celui de la page WEB qui référence l'applet. L'applet est active si la page WEB est active, elle est inactive si la page WEB est inactive.
Détruite	L'exécution de l'applet est terminée. Cette terminaison arrive quand l'applet est déchargée (« unloaded »), et ceci typiquement lorsque le navigateur est fermé. Certains navigateurs toutefois, opèrent un rechargement de l'applet lorsque l'utilisateur retourne à la page de l'applet.

Aux changements d'état de l'applet sont associées 4 méthodes définies dans la classe `Applet`: `init`, `start`, `stop` et `destroy`.

Ces 4 méthodes sont appelées automatiquement par la machine virtuelle à l'occasion d'un changement d'état. **Par défaut, ces 4 méthodes ne font rien ! Le concepteur de l'applet est libre de les redéfinir à son grès, selon ses besoins.** L'important est qu'il sache exactement quand ces méthodes sont appelées.

`init()`	Ce message est envoyé à l'applet dès que cette dernière est chargée et dispose de tout son environnement. Il est recommandé au programmeur de placer son code d'initialisation dans cette méthode plutôt que dans le constructeur. Notamment pour tout ce qui concerne l'accès au réseau (chargement d'images par exemple). En effet, le programmeur a la garantie que le message «`init()`» sera reçu une fois seulement que l'applet disposera de tout son environnement.
`start()`	Ce message est envoyé après le message «`init()`», ou dès que la page HTML devient à nouveau active. C'est par exemple dans cette méthode que l'on placera les instructions propres au démarrage de « threads ».
`stop()`	Ce message est envoyé dès que la page HTML devient inactive. En principe, le programmeur devrait exploiter cette méthode pour suspendre ou pour stopper certaines activités lancées à l'occasion du message «`start`», activités dont l'exécution est devenue inutile puisque l'utilisateur ne visionne plus la page HTML.
`destroy()`	Le navigateur peut être **fermé**, ou l'applet peut être rechargée («**reloading**»). Dans les deux cas, notre applet va mourir… On lui donne l'occasion de dicter ses dernières volontés dans la méthode «`destroy()`». Si l'applet se trouve initialement à l'état Active, le message «`stop()`» lui est d'abord envoyé. Cette dernière passe donc d'abord à l'état Inactive avant d'être détruite. Le programmeur peut placer dans la méthode «`destroy()`» d'éventuelles instructions de « nettoyage », utilisées par exemple pour libérer certaines ressources.

47.2.1 Un exemple : illustration du cycle de vie

Voici, pour illustrer les notions que nous venons de présenter, le code d'une petite applet sans prétention qui se contente d'afficher ses changements d'état.

```
import java.applet.Applet;
import java.awt.Graphics;

public class Applet2 extends Applet {
```

```
    private StringBuffer sb= new StringBuffer();

    public Applet2 () { sb.append ("Construction...");}

    public void init() {
       afficher("init... ");
    }

    public void start() {
       afficher("start... ");
    }

    public void stop() {
       afficher("stop... ");
    }

    public void destroy() {
       afficher("Destruction imminente...");
    }

    private void afficher(String mot) {
       sb.append(mot);
       repaint();
    }

    public void paint(Graphics g) {
        // Procédure appelée:
        // La première fois que l'applet est affichée,
        // A chaque fois que la surface de l'applet nécessite un rafraîchissement,
        // A chaque invocation du message "repaint()"

        //Dessine un rectangle autour de la zone de texte
        g.drawRect(0, 0, size().width - 1, size().height - 1);

        //Affiche le texte à l'intérieur du rectangle
        g.drawString(sb.toString(), 5, 15);
    }
}
```

Son exécution au moyen de «l'appletViewer» affichera ceci :

```
Construction…init…start…
```

Si la même applet est exécutée au moyen d'un navigateur, on verra s'afficher ceci :

```
Construction…init…start…stop…start… stop …start…
```

La paire « `stop…start…` » apparaîtra à chaque fois que la page HTML qui référence l'applet aura été désactivée, puis activée à nouveau.

Le texte `"Destruction imminente...",` affiché au moment de la fermeture du navigateur, n'aura pas le temps d'être visualisé, à moins d'avoir une machine particulièrement lente.., ce qui ne saurait être le cas, vous connaissant.

47.3 DESSINER DANS UNE APPLET, AJOUT DE COMPOSANTS « GUI »

Les applets, contrairement aux applications autonomes, n'ont pas besoin de créer de

fenêtre: l'applet s'exécute d'office dans la fenêtre du navigateur[1][3]. Toutefois, ceci n'empêche pas à une applet de créer une ou plusieurs fenêtres supplémentaires si le cœur lui en dit.

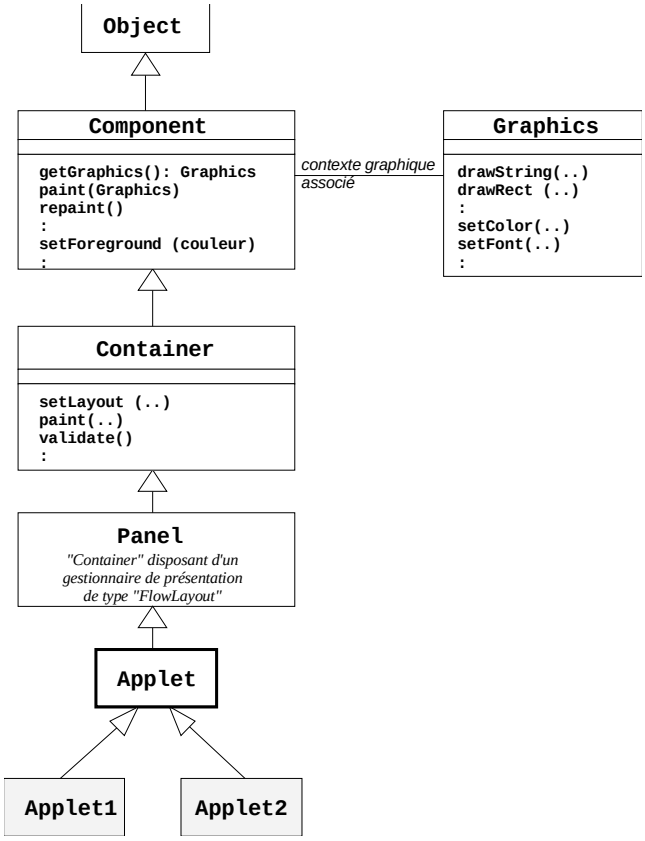
47.3.1 Les applets sont de riches héritières..

Une applet est réalisée par dérivation de la classe «**Applet**». Nous l'avons remarqué au travers de nos deux premiers exemples : «Applet1» et «Applet2» :¨

```
public class Applet1 extends Applet {..}
public class Applet2 extends Applet {..}
```

Voici la situation de nos deux applets au sein de la hiérarchie d'héritage:

Figure 2: La class «**Applet**» au sein de la hiérarchie d'héritage



1/ Une applet est un « **Component** »
En tant que tel, une applet est un composant graphique, qui peut interagir

1. [3] Attention à la portabilité ! les composants d'une applet (boutons, champs texte, ..) peuvent nécessiter un espace graphique dont la taille diffère d'un navigateur à l'autre. Il faut donc prévoir de la marge. Par ailleurs, il est vivement conseillé d'utiliser des gestionnaires de présentation flexibles, qui s'adaptent à la taille de la fenêtre, comme par exemple «BorderLayout» ou «GridBagLayout».

avec l'utilisateur.

Ainsi, l'applet est associée à un contexte graphique dont on peut spécifier la fonte, la couleur et dans lequel on peut dessiner au moyen de méthodes telles que «`drawString`», «`drawRect`», .. La classe «`Component`» met à disposition la méthode «`repaint()`» , qui efface la surface du composant et appelle la méthode «`paint(..)`» définie ou redéfinie pour ce composant.

Au niveau des événements, on pourra récupérer les actions de la souris, la pression des touches du clavier, le changement de focus, ..

2/ Une applet est un « **`Container`** »
En tant que tel, une applet est un composant graphique qui peut contenir d'autres composants : des boutons, des champs de saisie, etc.. Ces composants sont rajoutés au moyen des méthodes «`add(..)`».

Un «`Container`» propose par ailleurs une méthode «`paint(..)`», qui a pour effet de distribuer cet appel à tous les composants qu'il contient. Si cette méthode est redéfinie au niveau d'un Container, ne pas oublier de la terminer par «**`super.paint(g)`**» afin que tous les composants reçoivent à leur tour le message «`paint`».

Enfin, un «`Container`» propose la méthode «**`validate()`**», qui a pour effet d'afficher à nouveau tous ses composants. A utiliser après avoir opéré des adjonctions ou des suppressions de composants de manière dynamique alors que l'applet est déjà affichée.

3/ Une applet est un « **`Panel`** »
Un «`Panel`» n'est rien d'autre qu'un «`Container`» dont le gestionnaire de présentation a été pré-établi pour être de type «`FlowLayout`»: les composants sont alignés de gauche à droite au fur et à mesure de leur adjonction.

47.3.2 Dessiner dans une applet

En tant que «`Component`», une applet dispose d'un contexte graphique dont la taille de la surface a été définie au moment du chargement de l'applet (paramètres «`WIDTH`» et «`HEIGHT`» de la balise `<APPLET>`).

Pour dessiner, il suffira à l'applet de s'adresser à son contexte graphique au sein de la méthode **`paint`**, héritée de Applet, mais que l'on peut redéfinir:

```
public void paint (Graphics g) {
     super.paint(g);
     g.drawString ("Bonjour ", 10, 10) ;
     g.drawRect(10, 10, 100, 100);
     ..
}
```

La méthode «`paint`» est appelée automatiquement par la machine virtuelle la première fois que l'applet est affichée. Par la suite, cette méthode est appelée à chaque fois

que la surface graphique de l'applet nécessite un rafraîchissement (la page a été recouverte par une autre application), ou lorsque le programmeur lui-même appelle la méthode «**repaint**»

```
this.repaint();
```

L'exécution de la méthode «repaint» a pour effet:

1/ d'effacer la surface graphique du composant en utilisant la couleur de fond du composant,
2/ d'assigner la couleur de premier plan du composant à la couleur du contexte graphique,
3/ d'appeler la méthode «paint» pour redessiner l'applet.

Pour une plus grande efficacité, le programmeur peut définir la zone de la surface graphique à effacer:

```
this.repaint(x, y, largeurZone, hauteurZone);
```

47.4 LES THREADS ET LES APPLETS

Une activité interne exécute ses instructions concurremment au reste de l'application.

Remarquons en préliminaire que l'exécution d'une applet est un processus qui comporte déjà, au départ, plusieurs activités concurrentes. A ces activités viennent se rajouter les threads des activités internes définies par le programmeur lui-même.

En effet, l'exécution même d'une applet fait intervenir les activités suivantes :

- un thread pour l'AWT, responsable de l'interface avec l'utilisateur : affichage dans la fenêtre et gestion des événements ;

- un ou plusieurs threads (ça dépend ..), qui, placés sous le contrôle du navigateur, ont la responsabilité d'appeler les méthodes «init», «start», «stop» et «destroy». Dans le cas où le navigateur utilise plusieurs threads, ces derniers appartiennent au même « groupe de threads », ainsi que tous les threads créés en interne par les méthodes «init», «start», etc..

La définition d'une activité concurrente au sein d'une applet est chose courante. Notamment, il n'est pas rare d'utiliser un thread pour opérer des initialisations en arrière-plan. Par exemple, si l'applet doit charger un certain nombre d'images pour mettre en œuvre une animation, il y a tout intérêt à placer le code de chargement dans une activité parallèle. Le placer dans la méthode «init()» aurait pour conséquence de bloquer l'exécution de l'applet en attendant que les images aient été complètement chargées[14].

1. [4] Notons toutefois que les images au format GIF et JPEG sont chargées automatiquement par une tâche de fond que le programmeur n'a pas besoin de gérer.

Un modèle pour gérer le thread d'une applet

!! **Les activités internes d'une applet s'exécutent même quand l'applet n'est plus active** !! Après avoir démarré, une activité interne continue son exécution tant que la méthode «`run()`» n'est pas terminée, et ceci indépendamment du fait que l'applet soit active ou non.

Dans la plupart des cas, on libère les ressources occupées par une applet lorsque la page qui la contient n'est plus active.

L'idée consiste à exploiter la méthode «**`stop()`**» en écrivant les instructions qui entraînent la destruction des threads créés par l'applet. Ces threads seront créés à nouveau quand la page sera à nouveau active en plaçant le code correspondant dans la méthode «**`start()`**».

Dans tous les cas, les activités internes ne survivront pas à l'applet ! quand cette dernière sera détruite à l'occasion par exemple de la fermeture du navigateur, le message «`kill()`» sera envoyé au groupe de threads auquel elles appartiennent.

Considérons l'exemple d'une applet simulant un chronomètre et qui disposerait d'une tâche interne pour maintenir à jour le comptage des millisecondes.

Voici l'entête de notre classe:

```
class Chronometre extends Applet
                  implements Runnable {

// Variable d'instance
Thread activité;     // Activité interne

:
```

Méthode «**`start()`**» :

```
public void start() {
    activité = new Thread(this);
    activité.start();
}
```

Le thread est créé puis démarré à chaque fois que l'applet est activée.

Méthode « **`stop()`** »:

```
public void stop() {
    activité = null;
}
```

L'applet est désactivée, on indique au thread courant qu'il doit terminer son exécution.

Méthode «`run()`»:

```
public void run() {

     while (Thread.currentThread() == activité) {


          .. Code de l'activité ..

          try {Thread.sleep(100);}
          catch (InterruptedException e) {}
     }
}
```

Le message «`Thread.currentThread()`» retourne le thread courant, en train d'exécuter les instructions de «`run()`».

Le test de la boucle «`while`» n'est plus vérifié dans l'un ou l'autre des deux cas suivants :

1/ l'applet a été désactivée, la méthode «`stop()`» a donné la valeur «`null`» à la variable «`activité`».

2/ l'applet a été désactivée, puis réactivée aussitôt. L'ancienne activité, qui était endormie, n'a pas eu le temps de remarquer que la variable «`activité`» était passée à «`null`». Cette variable référence maintenant une nouvelle activité (créée et démarrée par «`start()`»). **Nous retrouvons ainsi avec deux activités internes qui exécutent concurremment la méthode «`run ()` » !!**

   A son réveil, l'ancienne activité constatera qu'elle n'est plus référencée par la variable «`activité`». Elle va donc terminer son exécution, et sera détruite sous peu. Ouf !

   La nouvelle activité continue normalement son exécution.

47.5 LA SÉCURITÉ ET LE DROIT DES APPLETS

Rappelons qu'une applet est une application téléchargée depuis n'importe quel point du réseau Internet. Cette possibilité soulève un grave problème de sécurité: le poste du client doit être protégé contre l'exécution d'une application qui violerait l'intégrité de son environnement.

Par exemple, il serait possible d'écrire une applet qui établit une liste de tous les fichiers « .exe » du disque dur local et envoie cette information à un site Internet intéressé.

Il serait encore possible d'écrire une applet qui installe un virus sur le disque local…

Le concept de sécurité est donc primordial, sans quoi personne n'accepterait de télécharger des applets.

Nous commençons par décrire les deux premiers niveaux de sécurité, offerts par le langage lui-même, puis nous présenterons le troisième niveau de sécurité, assuré cette fois-ci par les navigateurs.

Niveau 1 : la sécurité offerte par le langage Java

Le langage Java offre au départ un certain nombre de caractéristiques qui permettront d'exécuter les applets avec une certaine sécurité.

Ces caractéristiques empêchent principalement les applets d'avoir un accès direct à la mémoire. Cet accès direct permettrait aux applets :

- de détruire ou de modifier des données du client,
- de communiquer des données confidentielles du client,
- de détruire du matériel situé sur la machine du client,
- d'accaparer les ressources du client et de rendre ainsi sa machine inutilisable.

Voici certaines de ces caractéristiques, que le compilateur vérifiera:

- le langage Java n'utilise pas de pointeurs de manière explicite. Les objets sont accédés en les identifiant spécifiquement au moyen d'un nom. Cette absence de pointeurs ne permet pas d'accéder directement à certaines positions mémoire de la machine.

  Notamment, contrairement à C++, Java manipule les textes et les tableaux au moyen de structures de haut niveau sans avoir besoin de manipuler une quelconque arithmétique de pointeurs.

- l'accès aux éléments des tableaux est contrôlé en temps réel, pendant l'exécution du programme : un dépassement du tableau (ce qui reviendrait à un accès direct en mémoire) provoque aussitôt la levée d'une exception.

- le compilateur Java vérifie que toutes les coercitions de type sont légales. Rappelons que Java est un langage fortement typé.

Ainsi, la modification indirecte des références mémoire par le programmeur est impossible. Par ailleurs, l'environnement d'exécution vérifie que la coercition d'un objet vers une sous-classe est bien conforme à ce qui avait été prévu au niveau de la compilation.

Niveau 2 : contrôle de la conformité du « byte-code »

Le « niveau 1 » de sécurité est assuré par le compilateur.

Peux-t'on toujours se fier aux compilateurs ? évidemment non. Raison pour laquelle le code généré par le compilateur (le « byte-code ») sera une nouvelle fois vérifié avant d'être interprété.

Ainsi, à l'occasion du téléchargement d'une applet, le « **chargeur de classes**» (« class loader ») vérifie que les fichiers « `.class` », - qui contiennent le « byte-code » à interpréter -, sont bien **conformes** à la spécification du langage[15].

Le navigateur n'utilise qu'un seul « chargeur de classe ». Ce dernier est établi **par le navigateur**, au moment où le navigateur est lancé. Par la suite, ce chargeur ne peut pas ni modifié, ni surchargé. Notamment, les applets ne peuvent pas créer ou référencer leur propre chargeur.

Ce chargeur est indépendant de tout compilateur. Ainsi, il fonctionnera de manière identique et effectuera le même contrôle de conformité avec des « byte-code » issus de la compilation d'autres langages tels que Ada ou C++.

Une fois la vérification effectuée, il y a plusieurs possibilités. Les deux dernières permettent d'améliorer sensiblement les performances.

- interpréter le code instruction par instruction,
- associer l'interpréteur à un compilateur « à la volée », qui compile les instructions dans le code natif de la machine cliente lors de leur première exécution. Ce code natif étant exécuté lors des exécutions ultérieures.
- compiler le byte-code en créant du code natif (code de la machine cliente).

Niveau 3 : la sécurité offerte par les navigateurs

Les navigateurs interviennent pour le téléchargement et pour l'exécution des **applets**. Notre discours est donc centré sur les applets et non pas sur les applications autonomes (« Standalone Java applications »). En effet, du point de vue de la sécurité, ces deux types d'applications sont pris en charge de manière très différente.

- applications autonomes
  Une application autonome peut tout faire : lire et écrire sur le disque local, charger en mémoire des exécutables ou des librairies pré-compilées en code natif (« .dll »), ou encore communiquer avec des machines distantes via le réseau.
- applets
  Les applets ont **potentiellement** les mêmes droits..

  Toutefois, le navigateur contrôle ces accès et distingue deux types d'applets : les *applets dignes de confiance* et les autres..

  Le « **gestionnaire de sécurité** » (« Applet Security Manager »), appelé quelquefois le « **policier** .. » est un mécanisme toujours actif, qui assure le respect

1. [5] Notamment, il est vérifié:
qu'il n'y aura pas de « stack overflow »,
que les accès aux registres sont bien corrects,
que les paramètres des instructions sont corrects,
qu'il n'y pas de conversions de données illégales.

des restrictions d'accès pendant l'exécution même de l'applet.

Régler le niveau de confiance

Dans ce dernier cas, l'utilisateur peut spécifier le niveau de confiance en distinguant les « applets signées » (voir ci-après), des « applets non signées ». Dans les deux cas, l'utilisateur indiquera si les applets sont dignes de confiance ou si elles ne le sont pas. Si oui, il précisera un niveau de restrictions : élevé, moyen ou faible, niveau qui dépend évidemment d'un browser à l'autre.

- un **niveau élevé** de sécurité signifie que l'applet n'a aucun droit
- un **niveau faible** de sécurité signifie au contraire que l'applet a tous les droits
- le **niveau intermédiaire**, dit « **bac à sable** » (**sandbox**) permet à l'utilisateur de spécifier les objets placés dans le bac à sable et avec lesquels l'applet peut s'amuser..

Les applet signées

Une ***applet signée*** est une applet qui comporte l'identification de son auteur : celui-ci a accepté de confier son identité, laquelle est encryptée dans l'applet. Cette signature garantit que les données qui sont chargées sont bien celles que le client a demandées. L'utilisateur doit spécifier au navigateur la liste des signatures autorisées.

Attention toutefois.. est-ce que l'auteur en question est vraiment digne de confiance ?

Le gestionnaire de sécurité

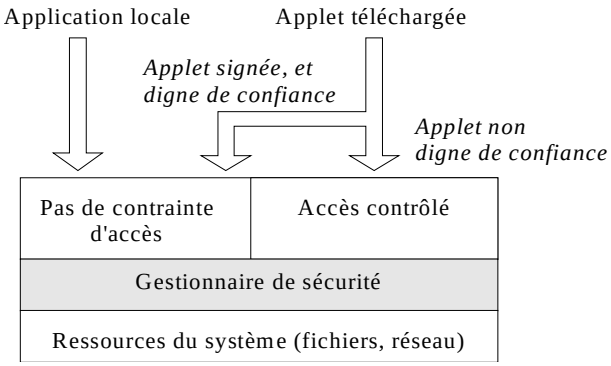
Voici quelques informations relatives au gestionnaire de sécurité que nous avons mentionné plus haut.

Par défaut, les applications autonomes s'exécutent sans gestionnaire de sécurité. Elles peuvent donc tout se permettre…[16]

Il n'existe qu'un seul gestionnaire de sécurité au niveau du navigateur. Ce dernier est spécifié et installé au moment où le navigateur est lancé. Par la suite, ce gestionnaire ne peut être ni modifié ni surchargé, et les applets ne peuvent pas non plus ni créer, ni référencer leur propre gestionnaire de sécurité.

1. [6] Si cela s'avère gênant, il est possible de modifier ce comportement par défaut en écrivant et en installant son propre gestionnaire de sécurité.

Figure 3: Le gestionnaire de sécurité



Dès que le gestionnaire de sécurité détecte une violation de la sécurité, il lève aussitôt une exception de type «`SecurityException`». Cette dernière peut être récupérée par l'applet qui doit se résoudre alors à faire « autrement », si elle le peut ..

Toute méthode a la possibilité d'interroger le gestionnaire de sécurité afin de savoir si telle ou telle opération est permise ou non (voir les méthodes «`checkXXX`» de la classe «`SecurityManager`»).

47.6 CE QUI EST INTERDIT AUX APPLETS

Ce qui suit s'adresse aux applets *non dignes de confiance*, qui ont donc des droits d'accès hautement limités.

Une applet ne peut pas :

1/ lire ou écrire dans des fichiers situés sur le disque local du client ;

2/ ouvrir une connexion Internet autrement qu'avec l'hôte depuis lequel l'applet a été téléchargée; pour contourner cet empêchement, il suffit que l'applet communique avec une *application autonome* située sur le serveur. Cette dernière, en tant qu'application autonome, aura la possibilité de se connecter sur d'autres sites.

3/ démarrer un exécutable situé sur la machine du client ;

4/ charger en mémoire des exécutables ou des librairies pré-compilées (**« `.dll` »** par exemple). Bien entendu, une applet peut utiliser son propre code, placé dans le répertoire d'hébergement du serveur, ainsi que les classes de l'API Java, situées dans l'environnement du navigateur ou de «l'appletViewer».

5/ définir des *méthodes natives*, c'est-à-dire des méthodes pré-compilées dans le langage de la machine client, et qui pourraient commander indirectement des accès aux ressources.

Ouvrir une connexion réseau

Ouvrir une connexion avec l'hôte d'hébergement et donc la seule possibilité.

Il peut s'agir en fait :

- de l'hôte d'où provient la page HTML, si le code de l'applet et la page HTML qui la référence sont situées au même endroit ;

- de l'hôte spécifié dans le paramètre optionnel « `codebase` » de la balise <APPLET>, qui indique où trouver les fichiers « `.class` » à télécharger.

47.7 Ce qui est permis aux applets

Soyons positifs et parlons un peu des permissions..

Accès aux propriétés du système

Les applets, qu'elles soient exécutées par le navigateur ou par « l'appletViewer », peuvent accéder à certaines propriétés du système sur lequel elles sont en train de s'exécuter.

Par exemple, on pourrait écrire :

```
String s = System.getProperty ("os.name") ;
```

Parmi ces propriétés, on trouve :

java.version	No de version de Java
java.vendor	Spécification du fournisseur de la machine Java
java.vendor.url	Son adresse URL
os.name	Nom du système d'exploitation
os.arch	Architecture du système d'exploitation
os.version	Version du système d'exploitation
file.separator	Séparateur de répertoires (p.ex : « `/` »)
path.separator	Par exemple : « `:` »
line.separator	Séparateur de ligne

Les applets peuvent consulter cette information, il n'existe pas de moyen pour les en empêcher.

Le programmeur n'a pas non plus les moyens de permettre aux applets d'accéder aux autres propriétés, comme par exemple :

java.class.path	Le « classpath » de Java
java.home	Le répertoire d'installation de Java
user.dir	Le répertoire courant de l'utilisateur
user.home	Le répertoire de base de l'utilisateur
user.name	Le nom de l'utilisateur

Comment sauvegarder des informations ?

Comme une applet n'a pas le droit d'accéder au disque local de la machine sur laquelle elle s'exécute, le seul moyen est d'opérer cette sauvegarde du côté serveur : une applet peut en effet lire ou écrire à partir de fichiers situés du côté serveur[17]

Qu'en est-il des applets chargées depuis le système de fichiers ?

Si les applets sont chargées depuis le réseau, elles sont sujettes aux restrictions d'accès que nous avons mentionnées.

Si au contraire l'applet réside sur le disque local du client, dans un répertoire appartenant au « CLASSPATH » du client, les restrictions d'accès sont beaucoup plus faibles.

D'une part, de telles applets ne sont pas passées par le « vérificateur de code ». D'autre part, elles peuvent:

- lire ou écrire dans des fichiers du disque local,
- charger des librairies (« `.dll` ») en mémoire vive,
- lancer des exécutables.

47.8 Comment manipuler les ressources WEB dans une applet ?

Tout en restant dans le cadre strict défini par la politique des droits d'accès, une applet Java a l'autorisation d'accéder à des ressources de nature très variées sur le réseau : des fichiers son, des images, des vidéos, ..

47.8.1 Identification d'une ressource (l'adresse «URL»)

Une ressource sera identifiée de manière non ambiguë par son adresse Internet que l'on appelle une « URL » (« Uniform Reference Location »).

Voici le format général d'une URL :

1. [7] Ceci peut être opéré soit par le biais de l'application serveur (qui a le droit d'accès, en tant qu'application autonome), soit par le biais d'un exécutable dont le code est installé du côté serveur (script PERL, CGI, ou autre), et qui serait démarré depuis l'applet, soit encore par biais d'une « servlet », lancée en même temps que l'applet par la page html.

http:	**//eig.unige.ch**	**:80**	**/sons/bip.mav**
Protocole	Identification du serveur(adr. IP) Machine.Sous-domaine.Domaine	Port (Service)	Ressource

Cette information indique au navigateur le nom du protocole à utiliser.

1/ Le **protocole**

Bien que le protocole standard s'appelle « HTTP » (« HyperText Transert Protocol »), d'autres protocoles sont utilisés sur Internet, comme par exemple:

- « FTP » (« File Transfer Protocol ») pour le transfert de fichiers ;
- « SMTP » (Simple Mail Transfer Protocol ») pour le courrier electronique.

2/ **L'identification du serveur (adresse IP)**
Cette information désigne une machine située sur le réseau Internet.

En principe, il s'agit d'une adresse numérique, codée actuellement sur 32 bits, et écrite sous la forme de 4 entiers de 0 à 255 séparés par un point. Cette adresse porte le nom « d'adresse Internet », ou encore « d'adresse IP ».

En général, par commodité, on affecte un nom logique à cette adresse, qui sera composé d'un nom de machine, d'un nom de sous-domaine et d'un nom de domaine.

La correspondance entre le nom logique et l'adresse IP est effectuée par un « serveur de noms » (« DNS », « Domain Name Server »), que l'on trouve au niveau de chaque domaine.

3/ **Le numéro de port**
Un serveur peut être caractérisé par plusieurs programmes auxquels il est possible de communiquer via le réseau. Ces différents programmes sont appelés des « **services** ».

Chaque service est identifié par un numéro de port.

Les serveurs offrent un service http affecté de manière normalisée au numéro **80**. Ainsi, la valeur 80 est prise par défaut si l'adresse URL ne spécifie aucun numéro de port : il s'agit donc d'un champ facultatif.

4/ **L'identification de la ressource**
Cette information précise le chemin d'accès de la ressource sur la machine.

47.8.2 Comment récupérer l'URL de l'applet ou de la page qui référence l'applet ?

Pour charger un fichier, une applet utilisera en général un « **URL relatif** ». Plutôt que de décrire exactement où se trouve le fichier, cet URL se basera sur le répertoire de base

d'où provient l'applet.

A cette fin, le système propose deux méthodes :

1/ la méthode «`getCodebase`» héritée de la classe Applet, qui récupère l'URL spécifiant le répertoire à partir duquel l'applet a été chargée, et que l'on appellera par la suite le répertoire « code-base » ;

2/ la méthode «`getDocumentBase`», qui récupère l'URL de la page HTML qui référence l'applet, et que l'on appellera par la suite le répertoire « document-base ».

A moins que la balise <APPLET> ne spécifie une valeur pour le paramètre « `codebase` », les URL retournés par ces deux méthodes sont identiques.

Pour les applets « non dignes de confiance », seul l'URL « codebase » sera utilisé : une telle applet ne pourra accéder qu'à ce répertoire et aux sous-répertoires de ce dernier.

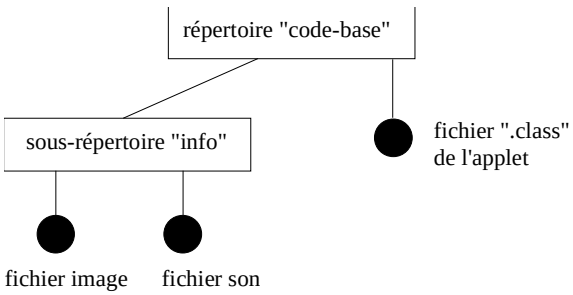
47.8.3 Exemple : chargement d'un fichier image ou d'un fichier son par une applet

Supposons un fichier image et un fichier son, tous deux situés du côté serveur, dans le répertoire qui héberge la page HTML.

Supposons que le nom de ces fichiers soit spécifié dans la balise <APPLET> de la page HTML, par le biais de deux paramètres « `image = …` » et « `son = …` ». La méthode «`getParameter`» permet d'obtenir la valeur de ces paramètres :

- String fichierImage = getParameter ("image") ;
- String fichierSon = getParameter ("son");

Supposons maintenant que ces deux fichiers se trouvent situés dans le sous-répertoire « `info` » du répertoire « code-base » :



Pour charger les deux fichiers, on écrira:

```
Image xx = getImage (getCodeBase(), "info/"+fichierImage);
AudioClip yy =getAudioClip (getCodeBase(),"info/"fichierSon);
```

En fait, le fichier image ne sera effectivement chargé que lorsque la commande «`drawImage (xx, ..)`» sera envoyé au contexte graphique de l'applet: `g.drawImage(xx, 0, 0, this);`

Le fichier son sera manipulé au moyen des méthodes «`play`», «`loop`» ou «`stop`»:

```
yy.play();
yy.stop();
yy.loop();// Jouer en boucle
```

47.9 LA BALISE <APPLET> ET LES PARAMÈTRES

Nous avons déjà introduit la balise <APPLET> du langage HTML. Nous allons maintenant étudier en détail l'environnement HTML des applets et voir notamment comment passer des paramètres à une applet.

Les paramètres permettent à l'utilisateur d'une applet d'influencer sur la façon dont cette dernière sera exécutée.

47.9.1 Spécifier les paramètres et leur valeur

Considérons l'exemple suivant :

```
<APPLET
CODE          = "xxxx.class"
WIDTH         = 450
HEIGHT        = 150
CODEBASE      = http://www.tcom.ch/applets/
ALIGN         = "LEFT"
HSPACE        = 10
VSPACE        = 10
ALT           = "impossible de charger l'applet"
>
<PARAM NAME = "nomDeLapplet" VALUE = "monNom">
<PARAM NAME = "autreParametreDeTypeEntier" VALUE = 10>
</APPLET>
```

Dans cette balise, seuls les paramètres «`CODE`», «`WIDTH`» et «`HEIGHT`» sont obligatoires, les autres sont facultatifs.

- **CODE**

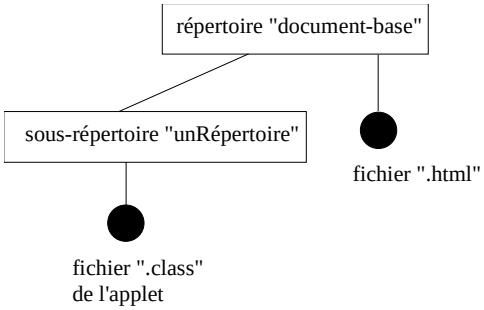
  Pour référencer le fichier « `.class` » de l'applet à exécuter

- **WIDTH**, **HEIGHT**

  Largeur et hauteur de la surface réservée à l'exécution de l'applet dans la fenêtre d'interprétation de la page HTML

- **CODEBASE**

  URL complet du répertoire contenant le fichier « `.class` » de l'applet à exécuter.
  Ce paramètre est obligatoire si cet URL est différent de l'URL qui héberge la page HTML.



  L'URL peut être spécifié **en absolu** (voir l'exemple plus haut) , ou **en relatif**. Dans ce dernier cas, l'URL est spécifié relativement à l'URL de la page HTML qui encapsule l'applet; très souvent, cet URL aura la forme suivante: "`unRépertoire/`", correspondant au schéma présenté ci-après.

- **ALIGN**

  Pour positionner l'applet par rapport à la ligne courante de la page HTML.

  Il est prévu de positionner l'applet par rapport au texte de la ligne courante. Le paramètre prendra alors l'une ou l'autre des valeurs suivantes :

  RIGHT : à droite du texte, LEFT : à sa gauche, TEXTTOP : le texte en haut, MIDDLE : le texte au milieu, BOTTOM : le texte en bas.

  Ou alors, l'applet sera positionnée par rapport au plus grand composant de la ligne courante. Il suffit de donner au paramètre l'une ou l'autre des valeurs suivantes :

  TOP : positionnée en haut, ABSMIDDLE : au milieu, ABSBOTTOM : positionnée en bas.

- **ALT**

  Pour désigner le texte qui sera affiché si l'applet ne peut pas être chargée

- **HSPACE** et **VSPACE**

  Pour désigner des espacements horizontal et vertical qui entourent la surface réservée à l'applet. Ces paramètres sont exprimés en pixels.

- la balise <**PARAM NAME** = "nomDeLapplet" **VALUE** = "monNom">

  Utilisée pour spécifier un paramètre « personnalisé », spécifié par son nom et sa valeur.

  De telles balises peuvent apparaître un nombre quelconque de fois. On peut ainsi spécifier autant de paramètres qu'on le désire.

  La valeur d'un paramètre peut appartenir à l'un ou l'autre des types suivants:

- **URL**
  - entier, réel
  - bouléen ("true" ou "false")
  - String

47.9.2 Lire la valeur des paramètres dans les instructions de l'applet

Pour lire la valeur des paramètres spécifiés dans la balise <APPLET>, on peut disposer de la méthode «`getParameter`»:

```
public String getParameter (String name)
```

Cette méthode retourne un «`String`» à convertir suivant le type du paramètre.

Voici quelques exemples:

```
String nomApplet   = getParameter ("nomDeLapplet");
int largeurSurface =

(Integer.valueOf(getParameter("WIDTH"))).intValue();
```

Comment aider l'utilisateur à utiliser correctement les paramètres ?

Les navigateurs peuvent utiliser les informations retournées par la méthode «`getParameterInfo`» pour aider les utilisateurs à spécifier la valeur des paramètres.

Pour exploiter cette possibilité, il suffit au programmeur d'écrire la méthode «`getParameterInfo`»:

Voici un exemple:

```
public String[][] getParameterInfo () {
     String [][] info = {
          // Nom du paramètre      Type            Description
          {"nomDeLapplet",    "String",      "nom de l'applet"},
          {"autreParametre", "int",          "bla bla bla"},
          :
     }
     return info;
}
```

Cette méthode doit retourner un tableau à deux dimensions, un tableau de tableaux de 3 chaînes de caractères.

47.9.3 Le paramètre « ARCHIVE »: les fichiers « JAR »

Si le code de l'applet est composé de plusieurs fichiers « `.class` », le chargement peut être long, très long…

Pour accélérer le processus, une première solution consiste à rassembler plusieurs classes dans un seul fichier. Ainsi, le nombre de connexions http sera réduit d'autant. Comme, le démarrage d'une connexion http peut nécessiter plusieurs secondes, l'économie de temps est conséquente.

Une deuxième solution est en fait une amélioration de la précédente : les fichiers « `.class` » sont rassemblés dans un ou dans plusieurs fichiers, qui sont eux-mêmes comprimés en s'appuyant sur le format « ZIP ». Il s'agit du format standard d'archivage utilisé par Java, que l'on appelle le format « **JAR** », introduit dans l'environnement JDK 1.1.

Les fichiers « JAR » devant être chargés pour exécuter l'applet sont spécifiés au moyen du paramètre « `ARCHIVE` » de la balise <APPLET> :

```
<APPLET

CODE                = "nomFichier.class"

ARCHIVE             = "fichierJAR1, fichierJAR2"
     :
</APPLET>
```

Evidemment, le ou les fichiers « JAR » doivent être préparés par le programmeur. A cette fin, l'environnement de développement Java fournit l'utilitaire « jar », prévu pour construire de tels fichiers.

Voici par exemple la commande qui permettra de rassembler toutes les classes (« `.class` ») et toutes les images (« `.gif` ») du répertoire courant dans un seul fichier compressé « `truc.jar` » :

```
jar cvf truc.zip *.class *.gif
```

47.10 POINTS DIVERS

Voici, pour terminer, une liste de remarques ou d'informations diverses, toutes relatives à l'utilisation des applets.

47.10.1 Affichage d'une ligne de statut

Les navigateurs offrent la possibilité aux applets d'afficher une ligne de statut.

Attention ! au cas où la page HTML référence plusieurs applets, cette ligne de statut sera partagée entre toutes.

La méthode «`showStatus`», à disposition, s'utilise ainsi :

showStatus ("Hello, patience, je charge un fichier… ») ;

47.10.2 Affichage d'une page HTML dans le navigateur

La méthode «`showDocument`», mise à disposition par la classe «`AppletContext`», demande au navigateur d'afficher une page HTML. Cette page sera spécifiée par son URL, et l'affichage aura lieu dans une fenêtre à spécifier également.

```
showDocument (unURL, fenêtreCible)
```

Concernant la fenêtre d'affichage, le deuxième paramètre peut prendre les valeurs suivantes :

`"_blank"`→ affichage dans une nouvelle fenêtre

`"_self"`→ affichage dans la fenêtre même qui contient l'applet

`"`*nomDuneFenêtre*`"`→ affichage dans une fenêtre à créer le cas échéant

…

47.10.3 Envoyer des messages à d'autres applets

Une page HTML peut référencer plusieurs applets qui s'exécuteront de manière concurrente.

Nous nous contenterons ici de mentionner le fait que toutes ces applets ont la possibilité de discuter entre elles en s'envoyant des messages.

A cette fin, la méthode «`getApplet`», de la classe «`AppletContext`», retourne une applet que l'on aura référencée pas son « nom ». Le nom d'une applet peut être spécifié au moyen du paramètre « NAME » de la balise <APPLET>.

Une fois que l'on dispose de la référence sur l'applet « cible », il suffit de lui envoyer n'importe quel message que cette dernière définit par le biais de ses méthodes.

```
Applet autreApplet =
     getAppletContext().getApplet("nomApplet") ;
autreApplet.m(..) ;
```

// Envoi d'un message à l'autre applet

47.10.4 Jouer des sons

A partir de la version 1.2 du « Java Development Kit », les applets et les applications autonomes ont la possibilité de manipuler des sons dans différents formats : mav, midi, ..

- la méthode «`getAudioClip`», héritée de Applet, permet de charger un fichier son ;
- les méthodes «`play`», «`loop`» et «`stop`», de la classe «`AudioClip`», permettent de manipuler le son.

47.10.5 Charger une applet, ça peut être long ..

Les classes de l'applet sont chargées au travers du réseau, ça peut être parfois très long..

Pour accélérer le temps de chargement de manière très efficace, on peut utiliser les fichiers d'archivage au format « JAR ». Voir les points 47.9.3, page 91 et 40.3, page 51 pour obtenir plus d'informations à ce sujet.

Mais on peut aussi, pour y remédier, faire illusion en utilisant une applet dont la clase principale (la sousclasse de «`Applet`») est très petite. Dès lors, le message de statut de l'applet s'affichera très vite, pendant qu'une tâche de fond (un « thread » écrit par nos soins), s'occupera de pré-charger[18] les classes dont l'application aura besoin.

1. [8] méthode «`loadClass`» de la classe «`ClassLoader`»

48 Annexe B - Liste des opérateurs avec priorité

Java

Au sein d'une même expression, des opérateurs de type «L» (Gauche) et de même priorité seront évalués de gauche à droite. C'est le cas de la plupart des opérateurs.

Les opérateurs de type «R» (Droite) et de même priorité, seront au contraire évalués de droite à gauche. C'est par exemple le cas de l'opérateur d'affection.

Si le programmeur désire intervenir sur l'ordre d'évaluation, il utilisera des parenthèses en créant ainsi des sous-expressions qui seront évaluées de manière prioritaire.

Niveau de priorité	Type	Opérateur	Type de l'opérande	Fonction
0	L	`.`		Opérateur d'accès : envoi de message, accès aux variables d'instance
	L	`(..)`	Expression	Appel de fonction, Création d'une sous-expression
	L	`[..]`	Expression	Accès aux éléments d'un tableau
1	R	`++, --`	Arithmétique	pré- ou -post- Incrémentation, Décrémentation
	R	`+, -`	Arithmétique	Signe plus, moins (opérateurs unaires)
	R	`~`	Entier	NOT binaire, bit à bit
		`!`	Booléen	NOT logique
2	R	`(..)`	type	Conversion de type (coercition, casting)
	R	**`new`**	Classe	Création d'un objet
3	L	`*, /, %`	Arithmétique	Opérateurs de multiplication, de division et modulo
4	L	`+, -`	Arithmétique	Opérateurs arithmétiques de soustraction et d'addition
	L	`+`	String	Concaténation de chaînes
5	L	`<<, >>`	Entier	Décalage bit à bit
	L	`>>>`	Entier	Décalage à droite sans extension de signe
6	L	`<, <=,>, >=`	Arithmétique	Opérateurs relationnels
	L	**`instanceof`**	Objet, Classe	Retourne un booléen indiquant si l'objet est l'instance d'une certaine classe
7	L	`==, !=`	types primitifs	Comparaison de valeur (égale, différent)

	L	==, !=	objets	Comparaison de références (égale, différent)
8	L	&	Entiers	AND bit à bit
	L	&	Booléens	AND bouléen (les 2 termes sont évalués)
9	L	^	Entiers	XOR bit à bit
	L	^	Booléens	XOR booléen
10	L	|	Entiers	OR bit à bit
	L	|	Booléens	OR booléen
11	L	&&	Booléens	AND logique conditionnel (un seul terme évalué)
12	L	||	Booléens	OU logique conditionnel (un seul terme évalué)
13	R	?:	Exp. bool., xx, xx	if arithmétique (opérateur ternaire)
14	R	=, *=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, |=, ^=	variable, xx	Opérateurs d'affectation et d'affectations composées

49 Annexe C – Création d'archives JAR

Le téléchargement de l'applet vers le client peut prendre du temps, beaucoup de temps, et peu provoquer par ailleurs un encombrement non négligeable du réseau.

Dans la mesure du possible, il faut donc tâcher de réduire la quantité d'informations à transférer, et trouver un moyen pour limiter la gêne du client.

Notons au préalable que le concept même des applets entraîne déjà à une réduction importante des informations à transmettre puisque l'exécution du code s'appuie sur les bibliothèques de le machine virtuelle installée dans le navigateur du client : seules les classes et les fichiers propres à l'application devront être téléchargés.

Pour réduire la quantité des informations à transmettre, la solution proposée par Sun consiste à compresser tous les fichiers à télécharger en un seul fichier que l'on appelle un fichier d'archivage. Ce dernier portera l'extension «.`jar`», avec un mode de compression s'appuyant sur le format zip.

49.1 Quelques considérations sur le temps de chargement des applets

L'engouement pour les applets a tourné court quand le public s'est rendu compte qu'il fallait parfois attendre une trentaine de secondes pour télécharger la page et le code de l'applet. La règle de base est donc de limiter la taille des applets à de petits programmes.

Comme le temps de chargement dépend directement du type de connexion, voici un tableau indiquant le temps de chargement minimum pour un fichier de 100 Kbytes, ainsi qu'une idée du volume des informations à transférer si on ne veut pas dépasser un temps de chargement de 5 sec (ce qui peut paraître raisonnable).

Débit	**Ligne modem** **56 Kbits**	**Ligne RNIS** **64 Kbits**	**Réseau Ethernet** **10 Mbits**	**Réseau Ethernet** **100 Mbits**
Temps pour 100 kbytes	15 sec	13 sec	0,08 sec	0,008 sec
Volume pour 5 sec	20 Kbytes	38 Kbytes	9,5 Mbytes	59,5 Mbytes

Ainsi, ce tableau montre que le volume occupé par la page html, le code de l'applet et les images éventuelles, ne devrait pas dépasser quelques dizaines de Kbytes dans un contexte internet avec des clients de caractéristiques diverses.

Dans un contexte intranet d'entreprise avec un réseau s'appuyant très souvent sur une couche Ethernet, le problème est bien différent. Le téléchargement d'une petite applet de 100 Kbytes sera quasiment instantané.

Notons par ailleurs que rares sont les applets dont la aille quelques dizaines de Kbytes. Le bytecode est en effet peu gourmand en mémoire : Sun avait prévu au départ d'utiliser Java pour des systèmes embarqués, et la réduction de la taille mémoire était à l'époque une priorité.

49.2 Compresser et agglomérer les fichiers à télécharger

Pour diminuer de façon notable le temps de chargement, la solution consiste à regrouper tous les composants (le bytecode, les images, les sons, ..) dans un seul fichier, et à compresser ce dernier.

L'utilisation d'un fichier compressé est doublement efficace :

- Moins d'informations à transmettre ;
- Et comme http coupe la connexion TCP/IP après chaque transfert de fichier, la réunion de toutes les classes en un seul fichier permet d'opérer une seule transaction http.

En contre partie, le navigateur devra décompresser le fichier. Mais ça n'est pas très grave si le client possède une machine puissante, comparée au débit du réseau.

Pour opérer cette compression, une première méthode consiste à utiliser l'utilitaire WinZip, mais il est préférable de comprimer avec l'outil de compression au format « **jar »**, proposé par Sun, et qui fait partie du JDK (depuis 1.1). En effet, ce dernier permet d'ajouter des informations d'authentification (la fameuse « signature électronique »).

49.3 Le format JAR

JAR est une archive zip selon la spécification PKWARE : il contient l'archive zip, plus éventuellement des fichiers de signatures électroniques et un fichier MANIFEST qui décrit la liste des fichiers de signatures éventuels présents dans l'archive.

Sa compatibilité avec ZIP lui permet d'être décompressé avec «`UnZip`».

49.4 Comment utiliser un JAR dans une applet ?

Pour utiliser dans une applet un jar plutôt qu'une classe, il suffit d'inclure le paramètre ARCHIVE :

```
<APPLET CODE = Xxx.class
```

```
ARCHIVE = "Yyy.jar"
WIDTH = 460,
   …
```

Il est possible d'inclure plusieurs archives, il suffit de les séparer par des virgules:

```
ARCHIVE = "xxx.jar, yyy.jar, zzz.jar"
```

L'archive n'a pas besoin d'être complète ! Si le navigateur n'y trouve pas ce qu'il cherche, il s'adresse au serveur en utilisant le paramètre CODEBASE, comme il le fait d'habitude.

49.5 SYNTAXE DE LA COMMANDE JAR

Operation	**Command**
Pour créer un fichier JAR	jar cf *jar-file input-file(s)*
Pour visionner le contenu d'un fichier JAR	jar tf *jar-file*
Pour extraire le contenu d'un fichier JAR	jar xf *jar-file*
Pour extraire des fichiers spécifiques d'un fichier JAR	jar xf *jar-file archived-file(s)*
Pour exécuter une application empaquetée dans un fichier JAR ("Jar executable") (à partir de la version JDK 1.2 --, nécessite un fichier manifest comportant l'entête Main-Class)	java -jar *app.jar*
Pour invoquer une applet empaquetée dans un fichier JAR	<applet code=*AppletClassName.class* archive="*JarFileName.jar*" width=*width* height=*height*> </applet>

Syntaxe

```
jar [options] [manifest] destination input-file
[input-files]
```

Les options de la commande jar

Un argument commençant par le caractère «@» peut être utilisé pour spécifier des paramètres supplémentaires, à raison d'un paramètre par ligne. Ces paramètres sont insérés dans la ligne de commande à la position correspondant à l'argument «@<nomDeFichier>».

- **c** Creates a new or empty archive on the standard output.

- **t** Lists the table of contents from standard output.

- **x *file*** Extracts all files, or just the named files, from standard input. If *file* is omitted, then all files are extracted; otherwise, only the specified file or files are extracted.

- **f** The second argument specifies a jar file to process. In the case of **c**reation, this refers to the name of the jar file to be created (instead of on stdout). For **t**able or **x**tract, the second argument identifies the jar file to be listed or extracted.

- **v** Generates verbose output on stderr.

- **m** Includes manifest information from specified pre-existing manifest file.

  Example use:

  jar cmf myManifestFile myJarFile *.class

  You can add special-purpose name-value attribute headers to the manifest file that aren't contained in the default manifest. Examples of such headers would be those for vendor information, version information, package sealing, and headers to make JAR-bundled applications executable.

- **0**Store only, without using ZIP compression.

- **M** Do not create a manifest file for the entries.

- **u** Update an existing JAR file by adding files or changing the manifest.

  For example,

  jar uf foo.jar foo.class

  → would add the file foo.class to the existing JAR file foo.jar, and

  jar umf manifest foo.jar

  → would update foo.jar's manifest with the information in manifest.

- **-C** Temporaily changes directories during execution of jar command while processing the next argument. For example,

  ```
  jar uf foo.jar -C classes bar.class
  ```

  would change to the classes directory and add the bar.class from that directory to foo.jar. The following command,

  jar uf foo.jar -C classes . -C bin xyz.class

  would change to the classes directory and add to foo.jar all f iles within the classes directory, but not the classes directory itself, and then change to the bin directory and add xyz.class to foo.jar.

If any of "files" is a directory, then that directory is processed recursively.

49.6 LES "JAR EXÉCUTABLES"

☺ Les archives JAR offrent en outre la possibilité de créer des exécutables Java que l'on peut exécuter par un simple double-clic !

Pour construire l'exécutable Xxx.jar

```
>javac Xxx.java
```

```
>jar cvf Xxx.jar listeDesClassesAInclure
```

```
>jar umf NomDuFichierManifest Xxx.jar
```

Contenu du fichier manifest (un fichier texte qui portera par exemple le nom «manifest.mf»)

```
Main-Class : Xxx
```

Pour lancer le jar exécutable

Un double-clic,

ou : `>java –jar Xxx.jar`

49.7 SIGNATURE D'UN FICHIER D'ARCHIVE

L'utilitaire «`jarsigner`», livré avec le JDK, permet de signer une applet : la signature est opérée directement sur le fichier «`jar`». Cette signature électronique permettra deux choses :

- D'une part d'authentifier l'auteur de l'applet (l'applet vient bien de telle ou telle personne, à qui on fait confiance) ;
- D'autre part de vérifier l'intégrité des données transmises sur le réseau (le code de l'applet n'a pas été remplacé en chemin par le code d'une autre applet douée

de mauvaises intentions).

50 Annexe D - Les principaux paquetages

Java

java.lang

Importé d'office dans les programmes ;

Contient les classes fondamentales qui accompagnent le langage de programmation : Strings, threads, exceptions, erreurs, ..

java.lang.reflect

Classes dédiées à l'introspection, pour connaître par exemple la liste des méthodes et des variables attachées à tel ou tel objet.

java.io

Entrées-sorties, traitées à la base sous la forme de flots séquentiels de bytes ou de caractères.

- à lecture et écriture dans des fichiers du disque dur local, ou situés à distance sur le réseau
- à communication d'informations avec des sockets
- à communication d'objets sérialisés (pour RMI)

java.net

Entrées/sorties avec des ressources URL, Connexions par Sockets et datagrammes

java.util

Collections d'objets (listes dynamiques, piles), Dates, String parsing, Générateur de nombres aléatoires

java.Math

Fonctions mathématiques

java.awt

Librairie graphique et création des interfaces utilisateurs : fenêtres, dialogues, composants interactifs,..

Swing (java 1.2)

Amélioration de l'interface utilisateur, plus grand choix de composants, de fontes, ..

Java 2D (java 1.2)

Grand choix de courbes, rotation, translation, ..

Java 3D (java 1.2)

JMF

Java Multimédia Framework : audio, video, synchronisation

Pour le réseau

- **java.sql**à jdbc
- **java.rmi**à RMI
- **javax.mail**à envoyer des mails
- Enterprise Java Beans

Annexe A Index