Présentation du langage *Java*

P. Albers

23 septembre 2024

1 Le langage Java

1.1 Les langages de programmation

Quelque soit le langage utilisé, le problème est de faire comprendre les instructions à la machine. La solution est de disposer d'un programme capable de *transformer* une séquence d'instructions de haut niveau (appelé *code source*), en une séquence d'instructions machine (appelé code binaire). Ce type de programme sont appelés *traducteurs*.

Selon ses caractéristiques, un programme *traducteur* est appelé un *assembleur*, un *compilateur* ou un *interpréteur*.
Les compilateurs et les assembleurs traduisent les programmes dans leur ensemble ; tout le programme doit être fourni au compilateur ou assembleur pour la traduction. Une fois la traduction effectuée, le résultat peut être soumis au processeur pour traitement. On parle dans les deux cas de *langage compilé*.
Les interpréteurs traduisent quant à eux les programmes instruction par instruction, et soumettent immédiatement chaque instruction traduite au processeur pour être traitée. On parle ici de *langage interprété*.

Java est un langage interprété. Il repose sur l'idée de traduire tout d'abord le programme source, non pas directement en langage *machine*, mais dans un pseudo langage universel constitué de *byte codes*. Il suffit que ce code en pseudo langage soit ensuite interprété par un programme spécifique sur la machine : on parle de *machine virtuelle*.

Le programme en *bytecodes Java* est indépendant de la plate-forme sur laquelle on l'exécute. C'est en effet à la machine virtuelle qui va l'interpréter de se charger de l'*adapter* à la plate-forme.

1.2 Présentation du langage *Java*

Le langage *Java* est né en 1991 dans les laboratoires de la société *Sun*. *Java* est un langage dit de haut-niveau qui permet d'écrire n'importe quel type d'applications, y compris des applications Web. Les concepteurs de ce langage ont adopté une syntaxe trés proche du langage C++, en se débarrassant des lourdeurs du langage C et en ajoutant la notion de machine virtuelle.

Java fait partie de la famille des langages objet. Il est à la fois simple et permet de développer des programmes sûrs et robustes, allant des réseaux aux applications temps réel. L'utilisation de machines virtuelles rend le langage portable sur les différentes plates-formes. Mais l'interprétation du code a comme principal inconvénient de rendre les applications Java un peu moins rapide que les applications écrite en langage natif comme en langage *C* par exemple, même si les technologies de machines virtuelles sont de plus en plus performantes.

Java met à la disposition du programme une grande collection de composants qui est groupé en bibliothéques composées de nombreuses classes et interfaces (API - *Application Programming*

Interface. Cela couvre des éléments de programmation de bases, mais aussi la gestion des accès aux bases de données, des mécanismes de sécurité ou encore les réseaux.

1.3 Syntaxe du langage Java

même s'il s'agit d'un langage objet, la syntaxe de Java est similaire au langage C. On y retrouve en particulier les mêmes opérateurs, les mêmes types primitifs et les mêmes instructions.

1.3.1 Commentaires

Les commentaires correspondent à une partie de code qui ne sert qu'au programmeur pour décrire brièvement tout ou partie du code. Les commentaires ne sont pas considérés par le compilateur ; ainsi, les commentaires ne se retrouvent pas dans le code machine.

Les commentaires en Java commence par les caractères $/*$ et se termine par les caractères $*/$. Un commentaire peut être sur plusieurs lignes.

On peut également mettre en commentaire tout le restant de la ligne en commençant par les caractères //.

Voici un exemple de code commenté :

```
{
   int i ; /* ceci est un commentaire */
   i = 9; /*  ceci est un autre
              commentaire  sur
              plusieurs lignes  */

   i = 10;  // ceci est encore un autre commentaire
   i = 11;
}
```

1.3.2 Variables et type primitifs

Une variable permet de stocker une valeur en mémoire de manière temporaire. Une variable est dèsignée dans le programme par un nom, et correspond physiquement en mémoire à une certaine quantité de mémoire. Le programme pourra non seulement connaître la valeur d'une variable, mais également la modifier en lui affectant une nouvelle valeur.

Le nom d'une variable peut contenir un nombre illimité de lettres ou de chiffres. Par convention, un nom de variable commencera nécessairement par une lettre minuscule. Il est important que les variables possèdent des noms évocateurs ; en particulier, il ne faut pas hésiter à utiliser plusieurs mots. Pour faciliter la lecture, il est d'usage que les différents mots constituant un nom de variables commencent par une majuscule (sauf le premier), comme par exemple dans le nom suivant :

```
unNomDeVariableTresLong
```

En Java, toute variable possède forcément un type qui lui est attribué à sa création. En particulier, une variable ne pourra changer de type en cours de programme. On peut considérer qu'un type permet de définir l'ensemble des valeurs possibles pour une variable.

On peut utiliser en Java soit les types primitifs qui sont prédéfinis, soit utiliser ces propres types en créant des *classes*[1].
Les types primitifs sont les suivants : *byte*, *short*, *int*, *long*, $float$, *double*, *boolean*, et *char*. Les quatre

1. La création de classes sera abordée ultérieurement.

premiers désignent des entiers ; la différence provient qu'ils sont associés à un espace mémoire plus ou moins important. Le programmeur pourra ainsi utiliser une variable de tel ou tel type en fonction des besoins de l'application :

— une variable de type *byte* est définie sur 8 bits pour des valeurs allant de -128 à 127,
— une variable *short* sur 16 bits de -32768 à 32767,
— une variable *int* sur 32 bits de -2^{31} à $2^{31}-1$
— et une variable *long* sera définie sur 64 bits pour des valeurs allant de -2^{63} à $2^{63}-1$.

Le type $float$ et *double* désigne des valeurs à virgule flottante. Les variables de type $float$ auront une précision sur 32 bits alors que les variables de type *double* auront une précision sur 64 bits.

Le type *boolean* définit le type booléen. Une variable de ce type aura pour valeur soit $true$, soit $false$.

Enfin le type *char* désigne l'ensemble des caractères *unicode* utilisables.

Lorsqu'une variable est créée, elle est initialisée par défaut comme indiquée dans le tableau suivant. Cependant, il est préférable d'initialiser de manière explicite toute variable, et de ne pas se fier à ces valeurs par défaut, qui sont source potentielle de problèmes.

type	byte	short	int	long	float	double	char	boolean
valeur par défaut	0	0	0	$0L$	$0.0f$	$0.0d$	$\backslash u0000$	$false$

1.3.3 Opérateurs

Les opérateurs permettent de faire des opérations sur les variables. Voici la liste des opérateurs possibles :

opérateurs arithmétiques

— l'opérateur +, qui définit l'addition pour les entiers et les nombres flottants,
— l'opérateur −, qui définit la soustraction pour les entiers et les nombres flottants,
— l'opérateur ∗, qui définit la multiplication pour les entiers et les nombres flottants,
— l'opérateur /, qui définit la division pour les entiers et les nombres flottants,
— l'opérateur %, qui définit le reste de la division entières uniquement pour les entiers,
— l'opérateur &, qui définit un *et bit à bit* entre deux entiers,
— l'opérateur |, qui définit un *ou inclusif bit à bit* entre deux entiers,
— et l'opérateur ∧, qui définit un *ou exclusif bit à bit* entre deux entiers,

opérateurs logiques

— l'opérateur == qui teste l'égalité entre deux valeurs (entiers, nombre flottant, caractère, booléen),
— l'opérateur != qui teste l'inégalité entre deux valeurs (entiers, nombre flottant, caractère, booléen),
— l'opérateur && qui fait un *et logique* entre deux booléens,
— l'opérateur || qui fait un *ou logique* entre deux booléens,
— et l'opérateur ! qui définit la négation d'un booléen.

opérateurs relationnels

— l'opérateur < qui teste si deux valeurs sont inférieures strictement (entier, nombre flottant, caractère),
— l'opérateur > qui teste si deux valeurs sont supérieures strictement (entier, nombre flottant, caractère),
— l'opérateur <= qui teste si deux valeurs sont inférieures (entier, nombre flottant, caractère),
— l'opérateur >= qui teste si deux valeurs sont supérieures (entier, nombre flottant, caractère),

— et l'opérateur *instanceof* qui teste si un objet est une instance d'une classe.

opérateurs de décalage

— l'opérateur << qui fait un décalage de bit pour un entier vers la gauche,
— l'opérateur >> qui fait un décalage de bit pour un entier vers la droite ; le bit le plus à gauche correspond au signe de l'entier,
— l'opérateur >>> qui fait un décalage de bit pour un entier vers la droite ; le bit le plus à gauche est mis systématiquement à 0,
— et l'opérateur $\sim$ qui correspond à l'inverse bit à bit d'un entier.

opérateurs d'affectation

— l'opérateur = qui permet d'affecter une valeur à une variable (variable de tout type),
— l'opérateur ++ qui permet d'incrémenter de 1 une valeur entière,
— l'opérateur −− qui permet décrémenter de 1 une valeur entière,
— l'opérateur −= qui est un raccourci pour retirer une valeur à une variable (entier et nombre flottant),
— l'opérateur ∗= qui est un raccourci pour multiplier une valeur à une variable (entier et nombre flottant),
— l'opérateur /= qui est un raccourci pour diviser une valeur à une variable (entier et nombre flottant),
— l'opérateur %= qui est un raccourci pour affecter le reste de la division entière d'une valeur entière,
— l'opérateur &= qui est un raccourci pour faire un *et bit à bit* avec une valeur entière,
— l'opérateur ∧= qui est un raccourci pour faire un *ou exclusif bit à bit* avec une valeur entière,
— l'opérateur |= qui est un raccourci pour faire un *ou inclusif bit à bit* avec une valeur entière,
— l'opérateur |= qui est un raccourci pour faire un *ou inclusif bit à bit* avec une valeur entière,
— l'opérateur <<= qui permet de décaler vers la gauche un entier,
— les opérateurs >>= et >>>= qui permettent de décalager vers la droite un entier.

opérateur de test

— et enfin l'opérateur ? : qui correspond à un raccourci de l'instruction *if-then-else*.

1.3.4 Priorités des opérateurs

Dans une expression, il se peut que plusieurs opérateurs apparaissent. Il existe des priorités entre les opérateurs de telle sorte qu'un opérateur ayant la plus grande priorité sera tout d'abord évalué. Si les opérateurs sont de même priorité, les expressions sont évaluées de gauche à droite, sauf pour les opérateurs d'affectation qui sont quant à eux évalués de droite à gauche.

Par exemple, l'opérateur ∗ a une priorité plus grande que l'opérateur +. C'est pourquoi l'expression $1 + 2 * 3$ aura comme résultat 7. Cette expression est donc syntaxiquement égale à $1 + (2 * 3)$.

De manière générale, l'utilisation des paranthéses est trés fortement recommandée, voire même indispensable pour éviter toute erreur très difficilement indécelable.

Le tableau suivant montre la précédence des différents opérateurs, du plus prioritaire au moins prioritaire. Les opérateurs se trouvant sur la même ligne ont la même priorité :

Opérateurs	**Précédence**
postfixés	$expr{+}{+}\ \ expr{-}{-}$
unaires	${+}{+}expr\ \ {-}{-}expr\ \ {+}expr\ \ {-}expr\ \ \sim\ \ !$
multiplicatifs	$*\ \ /\ \ \%$
additifs	$+\ \ -$
décalage	$<<\ \ >>\ \ >>>$
relationnels	$<\ \ >\ \ <=\ \ >=\ \ instanceof$
égalité	$==\ !=$
et bit à bit	&
ou exclusif bit à bit	$\wedge$
ou inclusif bit à bit	|
et logique	&&
ou logique	||
ternaire	? :
affectation	$=\ \ +=\ \ -=\ \ *=\ \ /=\ \ \%=\ \ \&=\ \ \wedge=\ \ \mid=\ \ <<=\ \ >>=\ \ >>>=$

1.3.5 Blocs d'instructions

Un bloc d'instructions correspond à un ensemble d'instructions où toutes les instructions vont être exécutées les unes à la suite des autres. Un bloc d'instructions commence par une accolade ouvrante et finit par une accolade fermante.

```
int i;
...
if(i==10)
{    // début du bloc "alors"
     ...
}    // fin du bloc "alors"
else
{    // début du bloc "sinon"
     ...
}    // fin du bloc "sinon"
```

Les variables qui sont déclarées dans un bloc d'instructions sont *locales* au bloc, c'est à dire qu'elles ne sont pas connues et sont donc inutilisables en dehors de ce bloc.

1.4 Les structures de contrôle

Les instructions d'un programme sont exécutées de haut en bas dans l'ordre d'apparition. Les structures de contrôle permettent d'organiser le code en bloc d'instructions.

1.4.1 Les conditions

L'instruction $si - alors$ exécute un bloc d'instructions en fonction d'une condition qui doit être vérifiée. La syntaxe de cette instruction est la suivante :

```
if( condition)  {
     instructions
}
```

Comme en langage C, il n'y a pas de mot clef pour $alors$. La condition est obligatoirement une expression de type booléen. Le bloc d'instructions peut être remplacé par une seule instruction. Si la condition est fausse, le bloc d'instructions est passé.

L'instruction *si – alors – sinon* exécute de la même manière un bloc d'instructions en fonction d'une condition qui doit être vérifiée ou non. La différence provient que l'on spécifie également le bloc d'instructions lorsque la condition n'est pas vérifiée. La syntaxe de cette instruction est la suivante :

```
if(condition)  {
     instructions1
} else {
     instructions2
}
```

Si la condition est vérifiée, le bloc d'instructions 1 est exécuté , sinon c'est le bloc d'instructions 2 qui est exécuté.

Comme pour l'instruction *si – alors*, le bloc d'instructions peut être substitué par une seule instruction. D'autre part, les instructions *si – alors – sinon* peuvent être imbriquées comme dans l'exemple suivant :

```
int i;
int j;
...
if(i==1) {
   j = 10;
} else {
   if(i==2) {
      j = 100;
   } else {
      if(i==3) {
         j = 100;
      } else {
         j=0;
      }
  }
}
```

On préférera cependant utiliser le code suivant :

```
int i;
int j;
...
if(i==1) {
   j = 10;
} else if(i==2) {
   j = 100;
} else if(i==3) {
   j = 100;
} else {
   j=0;
}
```

On peut remarquer que non seulement ce code est plus lisible, mais il limite également le nombre d'accolades et donc le nombre d'erreurs de syntaxe.

L'instruction de branchements multiples Lorsque le nombre de valeurs est important, l'instruction de branchements est parfois préférée à l'imbrication d'instructions $si-alors-sinon$. Elle ne fonctionne cependant que pour les entiers, les types énumérés et pour les variables de type *String*.

Voici la syntaxe de cette instruction :

```
int mois, nbJours;
...
switch(mois) {
   case 1 : nbJours=31; break;
   case 2 : nbJours=28; break;
   case 3 : nbJours=31; break;
   case 4 : nbJours=30; break;
   case 5 : nbJours=31; break;
   case 6 : nbJours=30; break;
   case 7 : nbJours=31; break;
   case 8 : nbJours=31; break;
   case 9 : nbJours=30; break;
   case 10: nbJours=31; break;
   case 11: nbJours=30; break;
   case 12: nbJours=31; break;
   default: nbJours=0;
}
```

En fonction de la valeur *mois*, les instructions correspondant à la valeur des étiquettes *case* sont exécutées jusqu'à rencontrer l'instruction *break*. Les instructions après le *switch* sont ensuite exécutées.

Attention de ne pas oublier de *break*, car les instructions du *case* suivant seraient également exécutées. En particulier, on pourrait optimiser le code précédent de la manière suivante :

```
int mois, nbJours;
...
switch(mois) {
   case 2 : nbJours=28; break;
   case 1 :
   case 3 :
   case 5 :
   case 7 :
   case 8 :
   case 10:
   case 12: nbJours=31; break;
   case 4 :
   case 6 :
   case 9 :
   case 11: nbJours=30; break;
   default: nbJours=0;
}
```

Les instructions de l'"étiquette $default$ sont exécutées lorsque la valeur *mois* à une autre valeur que celle listée. Cette étiquette $default$ est obligatoire, car il faut être à même de traiter toutes les valeurs possibles de *mois*.

1.4.2 Les boucles

Trois types de boucles sont possibles en Java : la boucle *pour*, la boucle *tant que* et la boucle *répéter tant que*. Les boucles sont utilisées pour signifier qu'un même bloc d'instructions est à exécuter un certain nombre de fois.

La boucle *tant que* a la syntaxe suivante :

```
while( condition ) {
     instructions
}
```

La *condition* est tout d'abord évaluée ; cette condition est nécessairement de type booléen. Si la condition n'est pas vérifiée, la boucle est passée et le programme continue. Si par contre elle est vérifiée, alors les instructions de la boucle sont exécutées. Une fois les instructions exécutées, la condition est de nouveau vérifiée, et ainsi de suite.

La boucle *répéter jusqu'à* possède la syntaxe suivante :

```
do {
     instructions
} while( condition );
```

à la différence de la boucle *tant que*, les instructions sont au moins exécutées une fois. Une fois les instructions exécutées, la condition est vérifiée. Si la contition est vérifiée, les instructions sont une nouvelle fois exécutée, *etc.*

La boucle *pour* a la syntaxe suivante :

```
for ( expression_d_initialisation ;
      expression_de_ contrôle ;
      expression_d_itération) {
  instructions_de_boucle
}
```

L'expression d'initialisation est tout d'abord exécutée. Si l'expression de contrôle est vérifiée, alors les instructions de la boucle sont exécutées. Puis l'expression d'itération est ensuite exécutée. Une nouvelle fois l'expression de contrôle est évaluée. Si cette derniére est vérifiée, alors les instructions de la boucle sont une nouvelle fois exécutées, et ainsi de suite.

On peut remarquer que l'expression d'initialisation n'est fait qu'une seule fois, alors que l'expression d'itération est exécutée après chaque fin d'exécution des instructions de la boucle.

On peut également remplacer un *for* par le *while* suivant :

```
expression_d_initialisation
while(expression_de_ contrôle){
   instructions_de_boucle
   expression_d_itération
}
```

Si la syntaxe du *pour* paraît tout d'abord assez primitive, elle offre en fait plus de possibilités. En l'occurrence, les expressions d'initialisation et d'itération peuvent être multiples et composées de plusieurs instructions séparées par des virgules. L'exemple suivant permet d'utiliser deux variables, dont l'une croit et l'autre décroit à chaque itération :

```
int i,j;
...
for(i=0, j=20;  i<j; i++, j--){
   ...
}
```

1.4.3 Choix du type de la boucle

On est souvent confronté au problème de savoir si l'on doit prendre une boucle *pour* ou une boucle *tant que* dans un sous-programme. Le *répéter tant que* n'est qu'une variante du *tant que*, où les instructions sont effectuées au moins une fois.

> Pour savoir si la boucle que l'on veut écrire doit être un *pour* ou un *tant que*, il suffit de se poser les questions suivantes :
>
> Est-ce que je connais précisément le nombre exact d'itérations de la boucle ?
> - Si oui, alors il s'agit d'un *pour*,
> - si non :
>
>   Est-ce qu'il faut faire au moins une fois les instructions de la boucle ?
>   - Si oui, alors il s'agit d'un *répéter tant que*,
>   - si non, il s'agit d'un *tant que*.

1.4.4 Conditions d'itération et conditions de sortie

Trouver les conditions d'itération d'un *tant que* (ou d'un *répéter tant que*) est parfois délicat. Il est souvent plus simple de considérer les conditions de sortie, et de prendre la négation de l'expression de sortie qui correspond à l'expression d'itération.

Prenons l'exemple d'un entier i qui varie dans une boucle, dont la condition de sortie serait lorsque sa valeur serait inférieure à 0 ou supérieure à 10, soit la condition : $i < 0 \ || \ i > 10$
Nous obtenons comme condition d'itération : $i >= 0$ && $i <= 10$, qui correspond à la négation de la condition de sortie.

1.4.5 Les branchements

Les instructions de branchements sont au nombre de trois : l'instruction *break*, l'instruction *continue* et l'instruction *return*.

Outre son utilisation dans l'instruction *switch*, l'instruction *break* peut être utilisée pour arrêter le bloc d'instructions d'une boucle. Dans l'exemple suivant, la boucle *pour* sera arrêtée lorsque la valeur de j sera égale à i.

```
int i,j;
...
for(i=0; i<10; i++) {
    ...
    if(j==i)
      break;
}
```

Mais ce code peut être remplacé la boucle *tant que* suivante :

```
int i,j;
...
i = 0;
while(i<10 && j!=i){
   ...
   i++;
}
```

La solution avec la boucle *tant que* est plus élégante et correspond à un meilleur style de programmation.

L'instruction *continue* permet d'interrompre partiellement les instructions d'une boucle. Soit le code suivant :

```
for(...) {
    bloc_d_instructions1
    if(condition)
          continue;
    bloc_d_instructions2
}
```

Si la *condition* est vérifiée, l'instruction *continue* permet de passer le bloc d'instructions 2, et de continuer la boucle pour une autre itération.

Les instructions *break* et *continue* ne font pas partie des bonnes pratiques de programmation. Il est donc fortement déconseillé de les utiliser.

L'instruction *return* permet quant à elle de retourner une valeur à n'importe quel moment dans une méthode. Il peut y avoir plusieurs *return* notamment avec l'utilisation des instructions *si-alors-sinon*.

1.5 Les tableaux

Un tableau permet de regrouper sous un même nom de variable un ensemble de valeurs de même type. En java, il est possible d'avoir des tableaux de n'importe quel type, que ce soit des types prédéfinis ou des types de n'importe quelle classe.

La déclaration d'un tableau se fait en Java en indiquant le type des éléments, suivi ou précédé par un crochet ouvrant et un crochet fermant. Voici par exemple la déclaration de deux tableaux d'entiers :

```
int tab1 [];
int [] tab2;
```

On peut indifféremment utiliser les deux notations, mais la deuxième forme sera préférée.

L'instanciation d'un tableau est faite par l'opérateur *new* en précisant le nombre d'éléments du tableau entre crochets. Le code suivant déclare une instance de tableau de 10 éléments :

```
int [] tab = new int [10];
```

La taille des tableaux est déterminée à l'instanciation ; elle ne peut pas être modifiée par la suite.

Deux variables de type tableau peuvent référencer une même instance de tableau. Dans l'exemple suivant, les variables *tab*1 et *tab*2 référencent la même instance de tableau. Le fait de modifier *tab*1 modifie en conséquence *tab*2, et inversement.

```
int [] tab1 = new int [10]
int [] tab2;
tab2 = tab1;
tab2[0] = 10;   //    <=>  tab1[0] = 10;
```

L'initialisation d'un tableau peut se faire directement en indiquant entre crochets les différentes valeurs séparées par des virgules.

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

Avec ce type d'initialisation, l'instance est automatiquement créée. La taille du tableau est également déterminée automatiquement en fonction du nombre d'éléments donnés.

Le nombre d'éléments de tout tableau peut être connue grâce au champ *length* comme suit :

```
int [] tab = {1,2,3,4,5};
int tailleTableau = tab.length;
```

Cet attribut permet d'avoir des fonctions génériques, notamment lors d'un passage d'un tableau en paramètre d'une méthode.

L'accès aux valeurs se fait valeur par valeur par l'intermédiaire d'un indice placé entre accollades. Le premier élément d'un tableau se situe à l'indice 0, et le dernier par conséquent à l'indice correspondant à la taille du tableau moins 1.

Voici l'exemple d'une méthode qui effectue la somme de toutes les valeurs d'un tableau d'entier :

```
int faireLaSommeDesValeurs(int [] tab) {
    int somme = 0;
    for(int i=0; i< tab.length; i++)
          somme += tab[i];
    return somme;
}
```

Lorsque l'on utilise un indice i dont la valeur est inférieure au 0 ou supérieure ou égale à la taille d'un tableau, l'exception $ArrayIndexOutOfBoundsException$ est déclenchée et le programme s'arrête [2].

Les valeurs par défaut d'une instance d'un tableau, lors de sa création, sont les suivantes :

— les valeurs des tableaux de type prédéfinis sont les valeurs des variables par défaut :

type	byte	short	int	long	float	double	char	boolean
valeur par défaut	0	0	0	$0L$	$0.0f$	$0.0d$	$\backslash u0000$	$false$

Ainsi, les valeurs initiales d'un tableau d'entier seront 0.

— les valeurs des tableaux d'une classe seront toutes égales à la valeur *null* : c'est à dire qu'il faudra par la suite créer une instance pour chaque élément du tableau.

L'égalité des valeurs d'un tableau ne peut pas être testé dans sa intégralité. Il faut balayer élément par élément pour tester l'égalité entre deux tableaux.

Il est possible de tester cependant l'égalité entre deux variables de type tableau, mais ce ne sont pas les valeurs qui sont testées, mais si les deux variables référencent la même instance. Ainsi dans le programme suivant, si $t1$ et $t2$ contiennent les mêmes valeurs, le test d'égalité $t1 == t2$ est faux, alors que le test $t1 == t3$ est vrai puisque $t1$ et $t3$ référencent la même instance de tableau.

2. Le mécanisme et le fonctionnement des exceptions seront abordès plus tard.

```
int [] t1 = {1,2,3,4};
int [] t2 = {1,2,3,4};
int [] t3 = t1;
// t1 == t3
System.out.println("t1 et t3 référencent la même instance");
// t1 != t2
System.out.println("t1 et t2 ne référencent pas la même instance");
```

Les tableaux multi-dimentionnels se déclarent en Java en précisant le nombre de dimensions par le nombre de crochets ouvrants et fermants. La taille des différentes dimensions est précisée lors de l'instanciation. On accéde aux différents éléments de ces tableaux multi-dimentionnels en précisant les indices pour chaque dimension entre crochets.

```
int [][] matrice = new int [3][5];   // 3 lignes et 5 colonnes
matrice[0][0] = 1;  ...; matrice[0][4] = 5;
matrice[1][0] = 6;  ...; matrice[1][4] = 10;
matrice[2][0] = 11; ...; matrice[2][4] = 15;

char [][][] vecteur3D = new char [10][10][10]; // vecteur à 3 dimensions
```

On peut de la même manière que les tableaux unidimensionnels initialiser les tableaux multi-dimentionnels. L'exemple suivant permet d'initialiser une matrice de 3 lignes et 4 colonnes :

```
int [][] matrice = {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
```

Toutes les colonnes d'une matrice n'ont pas forcément le même nombre d'éléments, comme dans l'exemple suivant :

```
int [][] matrice = {{1,2,3,4}, {5,6}, {7,8,9,10,11,12}};
```

On peut obtenir les longueurs des différentes dimensions de la manière suivante : *tab.length* qui donne le nombre de lignes de la matrice, *tab*[0].*length* qui donne la taille de la premiére colonne, *etc.*

1.6 Algorithmes de recherche

1.6.1 Parcourir un tableau

Le langage Java permet un moyen plus simple pour parcourir un tableau. Au lieu par exemple d'utiliser la boucle :

```
int [] tableau; ...
for(int i=0; i<tableau.lenght; i++){
    int element = tableau[i];
    System.out.println(element);
}
```

on peut avantageusement écrire :

```
int [] tableau; ...
for(int element: tableau){
    System.out.println(element);
}
```

Attention cependant lors de la modification des éléments du tableau, car dans l'exemple précédent *element* est une copie de *tableau*[*i*]. Le fait de modifier *element* ne modifiera donc pas *tableau*[*i*]. Ceci est valable pour tous les tableaux de type prédéfini (*int*, *float*, *char*, ...).

1.6.2 Chercher une valeur

Prenons l'exemple de la recherche de la présence d'une valeur dans un tableau. L'idée est d'arrêter la recherche dès que la valeur est trouvée ; en effet, ce n'est pas la peine de continuer la recherche. Nous avons alors deux cas de sorties : soit la valeur n'est pas présente auquel cas le tableau est parcouru dans sa totalité, soit la valeur est présente et la boucle s'arrête avant le parcours complet du tableau. Ne sachant pas à l'avance le nombre d'itérations possible de la boucle, nous choisissons une boucle *tant que*.
Voici la condition de sortie de la boucle, faisant intervenir un indice i initialisé à 0 et qui permet de balayer les différents éléments du tableau :

$$(i >= nombreDElementsDuTableau)||(tableau[i] == elementRecherche)$$

La condition d'itération est la négation de cette condition de sortie, ce qui donne le code suivant :

```
int [] tab = {1,2,3,4,5,6,7,8,9,10};
int valeurRecherchee = 5;
int i =0;
while(i<10 && tab[i]!=valeurRecherchee) {
     i++;
}
if(i==10){
     System.out.println("la valeur n'est pas présente");
} else {
     System.out.println("la valeur est présente");
}
```

1.6.3 Copie d'un tableau

Copier ou dupliquer un tableau nécessite de le parcourir dans son intégralité et copier les éléments un par un. Les deux tableaux doivent impérativement avoir la même taille.

```
int [] tab = {1,2,3,4,5,6,7,8,9,10};
int [] copie = new int [tab.length];
for(int i=0; i<tab.length; i++) {
    copie[i] = tab[i];
}
```

2 La programmation objet

2.1 Classes, attributs et méthodes

2.1.1 Définitions

Dans le monde réel, nous pouvons trouver de nombreux objets qui ont tous leurs propres caractériques. Un objet a été créé à un moment donné ; il peut rendre des services et finit par être détruit. En programmation informatique, il en est de même.

Certains objets possèdent la même structure et le même comportement. On peut alors modéliser ces objets en prenant en compte leurs caractéristiques communes et ainsi les classifier.

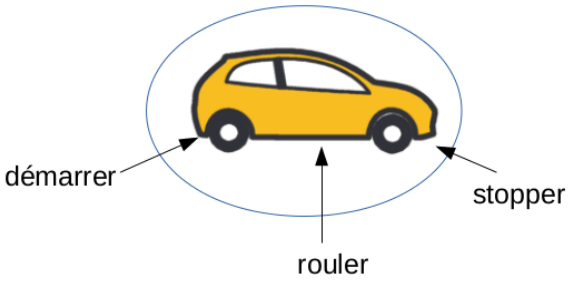
En programmation objet, une **classe** correspond à la déclaration d'une entité associant à la fois les données qui la caractérise et l'utilisation que l'on peut en faire.

On peut voir ces classes comme un cahier des charges ou une définition d'un objet. On assimile ainsi une classe à un modéle ou à un plan d'architecte. La définition d'une classe va ensuite servir à

construire des objets informatiques. Comme dans le monde réel, les objets d'un programme seront créés, rendront des services et seront détruits lorsqu'ils n'auront plus d'utilité.

Tous les objets créés auront leurs propres caractéristiques qui pourront se modifier au fil du programme. Par exemple, un objet de type *Voiture* pourrait avoir comme caractéristique son nom, sa marque, sa couleur ou encore sa cylindrée. Ces caractéristiques seront modélisées à l'aide de variables que l'on nomme en approche orientée objet des **attributs**. Tout objet aura ses propres attributs.

Un objet pourra par ailleurs réaliser un certain nombre d'actions qui représentent le *comportement* de l'objet. Ces actions seront modélisées à l'aide de fonctions, appelées **méthodes** en orienté objet. Par exemple, les méthodes possibles pour un objet *Voiture* pourraient être *démarrer*, *rouler* ou encore *stopper*.



Une classe se définit par conséquent comme un type abstrait de données caractérisé par des propriétés communes d'un objet (attributs et méthodes). C'est pourquoi, dans la suite de ce cours, nous confondrons la notion de type et la notion de classe.

Les classes sont généralement organisées en terme de hiérarchie, permettant ainsi une meilleure réutilisation du code. On identifie des liens particuliers entre les classes comme l'association, l'agrégation, la composition ou encore l'héritage. Ces notions sont les quatre principes de bases de la programmation orientée objet.

D'autre part, les attributs d'une classe pourraient être plus ou moins visibles vis à vis d'autres classes, permettant ainsi d'avoir un certain contrôle sur les propriétés d'un objet. On parle de *modificateurs d'accès* : quatre modificateurs différents sont présents en Java : *public*, *protected*, *private* et le modificateur par défaut.

2.1.2 Déclaration

En Java, une classe est définie par son nom de la manière suivante :

```
class Voiture {
        // mes attributs:
        ...
        // mes méthodes:
        ...
}
```

Une classe doit être obligatoirement déclarée dans un fichier d'extension *.java* et du même nom que la classe. La classe ci-dessus devra donc se trouver dans le fichier *Voiture.java*.

La convention *Java* demande à ce que tout nom de classe commence nécessairement par une majuscule. Comme pour les variables, si le nom d'une classe comporte plusieurs mots, aucun espace ne doit se trouver dans le nom, et les mots suivants doivent également commencer par une majuscule.

```
class UneClasseAvecUnLongNom {
        ...
}
```

2.1.3 La classe *String*

En Java, le type représentant les chaînes de caractères n'est pas un type primitif, au même titre que les entiers, les booléens ou encore les flottants. Par contre, il existe une classe qui définit le type *chaîne de caractère* : le type *String*.

On peut définir une chaîne de caractères comme une séquence de caractères. La manière la plus simple de déclarer une chaîne de caractères est la suivante :

```
String maPremiereChaine = "hello world !";
```

En Java, les variables de type *String* sont immuables, c'est à dire qu'elles ne peuvent pas changer de valeurs. En d'autres mots, il est impossible de modifier une variable *String* une fois qu'elle a été créée.

Nous devrons nous contenter de créer d'autres variables *String*, si nous voulons avoir une valeur modifiée. Il est possible en particulier de concaténer [3] deux chaînes de caractères pour en former une troisième, notamment avec l'utilisation de l'opérateur $'+'$. On peut aussi ajouter dans une chaîne des variables de type prédéfini comme dans l'exemple suivant :

```
String uneChaine = "bon" + "jour";
int a = 10;
String valeur = "la valeur de a vaut" + a;
```

L'affichage d'une chaîne de caractères se fait à l'aide de la méthode *System.out.println* de la manière suivante :

```
System.out.println("Hello world !");
int a = 10;
System.out.println("la valeur de a vaut=" + a);
```

Il est parfois nécessaire de convertir un nombre en chaîne de caractères. Plusieurs manières s'offrent à nous :

1. on peut concaténer avec une chaîne vide, car en effet une chaîne ne peut pas être initialisée uniquement avec un nombre :

   ```
   int a = 10;
   String valeur = "" + a;
   ```

2. on peut également utiliser la méthode *valueOf* de la classe *String* :

   ```
   int a = 10; String v1 = String.valueOf(a);
   float f = 20.; String v2 = String.valueOf(f);
   boolean b = true; String v3 = String.valueOf(b);
   ```

3. on peut enfin utiliser une autre classe, comme la classe *Integer* qui propose toutes les fonctionnalités autour des entiers [4] :

3. *i.e.* de mettre bout à bout
4. La plupart des classes possèdent une méthode *toString* qui renvoie une chaîne de caractères comprenant les valeurs de l'objet.

```
int a = 10; String valeur = Integer.toString(a);
float f = 20.; String v2 = Float.valueOf(f);
boolean b = true; String v3 = Boolean.valueOf(b);
```

Voici la liste des principales méthodes que proposent la classe *String*. Le restant des méthodes peut être vu dans l'API Java :

char	charAt(int i)	renvoie le $i^{\text{éme}}$ caractère de la chaîne. Le premier caractère de la chaîne a pour indice 0
int	compareTo(String s)	compare lexicographiquement une chaîne avec la chaîne s. Le résultat vaut 0 si les deux chaînes sont égales, une valeur positive si s se trouve avant la chaîne et une valeur négative dans le cas contraire
String	concat(String s)	retourne une nouvelle chaîne qui correspond à la concaténation de la chaîne avec s
boolean	isEmpty()	retourne *true* si la chaîne est une chaîne vide, c'est à dire une chaîne sans aucun caractère
int	length()	retourne le nombre de caractères de la chaîne
String	replace(char c1, char c2)	retourne une nouvelle chaîne de caractères en remplaçant toutes les occurrences du caractère $c1$ par le caractère $c2$
boolean	startsWith(String s)	retourne *true* si la chaîne commence par la chaîne s
String	subString(int i)	retourne une nouvelle chaîne qui est une partie de la chaîne à partir du $i^{\text{ème}}$ caractère. Le premier caractère à pour indice 0
String	toLowerCase()	retourne une nouvelle chaîne ne contenant que des minuscules. Les caractères qui ne sont pas des lettres restent inchangés
String	toUpperCase()	retourne une nouvelle chaîne ne contenant que des majuscules. Les caractères qui ne sont pas des lettres restent inchangés

2.2 Instance d'une classe

2.2.1 Définition

Une fois qu'une classe est définie, il est alors possible de créer des objets qui seront utilisés dans le programme. Ces objets auront des caractéristiques dont les valeurs seront stockés en mémoire. Cela implique que chaque objet possède une place mémoire unique qui lui est propre.

Dans le monde réel, un objet peut être désigné par plusieurs noms ; en programmation il en est de même. Dans un code, l'accès à tout objet ne peut se faire qu'à l'aide de variables.

2.2.2 Déclaration d'une variable

Lorsqu'une variable est déclarée, elle ne référence aucune instance. Par défaut, toute variable d'un objet aura la valeur *null*.

Si par mégarde on essaie d'appliquer une méthode sur une variable qui ne référence pas une instance, une exception est levée à l'exécution : $nullPointerException$. La plupart du temps, ce problème provient du fait que le programmeur a oublié d'affecter une instance à une variable. Il s'agit d'une erreur de programmation, et non pas d'une erreur de syntaxe. C'est pourquoi, ce genre d'erreurs n'est pas détéctée à la compilation, mais bien à l'exécution.

```
String s;  // déclaration d'une variable de type String
s.toUpperCase(); // déclenchement à l'exécution de l'exception :
                 // "nullPointerException"
```

2.2.3 Création d'une instance

La création d'un objet s'appelle, en programmation objet, une **instanciation**. Une instance est créée en Java à l'aide de l'opérateur **new** de la manière suivante :

```
Voiture v;         // déclaration d'une variable de type Voiture
v = new Voiture(); // création d'un objet voiture
```

On peut voir que le nom de la classe est rappelé dans la création d'instances.

Une variable d'un type particulier désigne n'importe quel objet de ce type. Ainsi on peut avoir plusieurs variables qui correspondent à un même objet. Á la suite du code suivant, les variables v et $v2$ correspondent à la même instance de l'objet $Voiture$:

```
Voiture v;         // déclaration d'une variable de type Voiture
Voiture v2;        // déclaration d'une deuxiéme variable de type Voiture
v = new Voiture(); // création d'un objet voiture
v2 = v;            // v et v2 désignent le même objet
```

Dans l'exemple précédent, comme v et $v2$ désignent la même voiture, le fait de modifier v, modifie en conséquence $v2$. En Java, lorsque plus aucune variable ne référence un objet, alors ce dernier est détruit et l'espace mémoire est réutilisé pour d'autres instances.

2.3 Attributs d'une classe

Comme vu précédemment, les caractéristiques d'un objet se modélisent à l'aide de variables que l'on nomme des **attributs**. Un peu comme dans les jeux de construction, un objet est fabriqué à l'aide d'autres objets. Ainsi, une voiture sera constituée d'un moteur, de fauteuil, d'un volant, *etc.*

Un attribut en Java se déclare à l'aide d'une variable d'un certain type. La classe $Voiture$ pourrait avoir la déclaration suivante :

```
class Voiture {
   Moteur monMoteur;
   Volant monVolant;
   Fauteuil fauteuilConducteur;
   Fauteuil fauteuilPassager;
   ...
}
```

2.3.1 Portée des variables et des attributs

À tout moment dans un programme, il est possible en Java de déclarer une variable. Cette variable est utilisable uniquement dans le bloc où elle est déclarée. On parle de la **portée** de la variable.

Par extension, les attributs d'une classe, étant déclarés dans le bloc principal de la classe, sont accessibles dans la totalité de la classe, y compris dans les méthodes de la classe. Ainsi, dans l'exemple suivant, la variable *monMoteur* est utilisable dans toutes les méthodes de la classe [5] :

```
class Voiture {
   Moteur monMoteur;
   Volant monVolant;
   Fauteuil fauteuilConducteur;
   Fauteuil fauteuilPassager;

   void demarrer(){
       monMoteur.mettreEnMarche();
       ...
   }
}
```

2.3.2 Le mot clef *this*

Le langage Java ne nous autorise pas à déclarer deux variables différentes de même nom. Il paraît évident que si cela était possible, l'ordinateur ne serait plus capable de différencier quel objet serait référencé.

Par contre, cette interdiction n'est applicable qu'aux variables qui seraient déclarées dans une même méthode. Il est possible en Java de déclarer des variables dans les méthodes qui auraient le même nom que les attributs de la classe.

Dans l'exemple suivant, la variable i est déclarée trois fois. La première fois, la déclaration de i correspond à un attribut de la classe ; la variable i qui apparaît dans la méthode $fonction1$ référence cet attribut. Par contre, la variable i qui apparaît dans la méthode $fonction2$ correspond à la variable locale de cette méthode ; *idem* pour la méthode $fonction3$.

```
class Test {
    int i;                   // première déclaration de "i"
    void fonction1(){
        i=0;
    }
    void fonction2(){
        int i=1;            // deuxième déclaration de "i"
        System.out.println(i);
    }
    void fonction3(int i){  // troisième déclaration de "i"
        i=1;
        System.out.println(i);
    }
}
```

Il se peut que l'on veuille référencer à la fois une variable locale et un attribut qui auraient le même nom. Pour les différencier, le langage Java nous met à notre disposition le mot clef *this*. L'utilisation de *this* devant le nom de la variable indique qu'il s'agit systématiquement de l'attribut. Dans l'exemple suivant, la méthode $fonction1$ prend en paramètre la valeur d'initialisation de l'attribut i :

5. Dans cet exemple, on appelle la méthode *mettreEnMarche* de l'objet *monMoteur*.

```
class Test {
    int i;                  // déclaration de l'attribut "i"
    void fonction1(int i){ // déclaration de la variable locale "i"
        this.i=i;           // l'attribut "i" prend la valeur de la variable locale "i"
    }
}
```

Pour prendre de bonnes habitudes et pour éviter toute erreur de programmation, nous vous recommandons de faire précéder systématiquement les attributs d'une classe par *this*, quelque soit la méthode.

2.3.3 Initialisation des attributs et constructeurs

Comme vu précédemment dans la section 2.2.3, la déclaration d'une variable d'un certain type ne correspond pas à la création d'une instance de ce type. Il est donc nécessaire de créer les objets correpondant aux attributs de l'objet que l'on est en train de créer. On peut voir cela comme une création en cascade.

Le mieux est d'utiliser une méthode qui permet d'initialiser toutes les variables d'une classe lors de sa création. Java met à notre disposition les **constructeurs** :

- Les constructeurs sont des méthodes particulières des classes et possèdent le même nom que leur classe. Ils ne **retournent aucune valeur** ; leur définition ne mentionne pas même le type *void*.
- Il peut y avoir plusieurs constructeurs pour une seule et même classe ; il suffit pour cela de les différencier par des paramètres différents, que ce soit par leur nombre et par leur type (voir section 2.4.1).
- Ces constructeurs sont appelés **implicitement** lors de la création des instances. On ne peut pas par ailleurs appeler un constructeur explicitement.

Dans l'exemple suivant, le constructeur de la classe $Voiture$ créera les instances de ses différents attributs ($monMoteur$, $monVolant$, ...) :

```
class Voiture {
   Moteur monMoteur;
   Volant monVolant;
   Fauteuil fauteuilConducteur;
   Fauteuil fauteuilPassager;

   Voiture(){
       monMoteur = new Moteur();
       monVolant = new Volant();
       fauteuilConducteur = new Fauteuil();
       fauteuilPassager = new Fauteuil();
       ...
   }
   void demarrer(){
       monMoteur.mettreEnMarche();
       ...
   }
}
```

L'appel au constructeur est réalisé de manière implicite juste après la création d'une instance, comme présenté dans l'exemple suivant :

```
class Test {
   public static void main(String [] args){
       Voiture v;         // déclaration d'une variable de type 'Voiture'.
       v = new Voiture(); // création d'une instance et appel au constructeur
                          //  de la classe 'Voiture'.
       v.demarrer();
   }
}
```

2.4 Méthodes et accesseurs

2.4.1 Déclaration des méthodes

En Java, les méthodes sont obligatoirement déclarées dans une classe. Elles possèdent un certain nombre de paramètres et peuvent renvoyer ou non une valeur.

```
TypeDeRetour nomDeLaMethode(Type1 parametre1, Type2 parametre2) {
      instructions
}
```

Le type de retour spécifie le type de la valeur retournée. En Java, si aucun paramètre n'est nécessaire, les parenthèses devront être néanmoins présentes.

Pour renvoyer une valeur, l'instruction *return* devra être utilisée. Dès que le programme rencontre cette instruction, la méthode s'arrête et le programme reprend le code à l'endroit où la méthode a été appelée. Une méthode peut contenir plusieurs instructions *return* ; toutefois la méthode s'arrêtera à la première instruction *return* rencontrée.

Il est possible de déclarer dans une même classe, deux méthodes ayant le même nom. Il faut cependant que les deux méthodes se différencient par leur nombre de paramètres et/ou par le type des arguments. En programmation orientée objet, on appelle cela la *surcharge* des méthodes.

Comme pour les attributs, on peut différencier l'accès aux méthodes en spécifiant un modificateur de visibilité : *public*, *protected*, *private* ou le modificateur par défaut.

2.4.2 Appel d'une méthode

L'appel à une méthode devra respecter, dans tous les cas, le nombre et le type des paramètres, ainsi que le type de la valeur retournée.

Lorsque l'on veut appeler une méthode depuis une autre méthode de la même classe, il suffit de l'appeler directement. Dans un premier temps, nous utiliserons systématiquement le mot clef *this*, comme pour les attributs (ce mot clef est optionnel). Voici un exemple d'appel :

```
class Test {
    void fonction1(){
        ...
    }
    int fonction2(int a, int b)){
        ...
    }
    void fonction3(){
        this.fonction1();
        int i= this.fonction2( 1,2 );
    }
}
```

Lorsque l'on veut appeler une méthode depuis une méthode d'une autre classe, **il faut spécifier nécessairement l'instance sur laquelle on l'applique**. Le code suivant appelle la méthode *length* de la classe *String* ; cette méthode n'a besoin d'aucun argument et renvoie un entier. La méthode *concat* quant à elle prend une autre chaîne en paramètre et renvoie une nouvelle chaîne de caractères :

```
class Test {
    void fonction(){
        String s = "hello";
        int i = s.length();
        String s2 = s.concat(" world!");
    }
}
```

2.4.3 Les accesseurs d'une classe

Comme tout objet, un attribut sera désigné dans la classe par une variable. Cet attribut sera connu dans l'ensemble de la classe, mais pas à l'extérieur. On parle en programmaton objet d'**encapsulation** des données.

L'encapsulation des données est basée sur le principe d'abstraction : un objet n'est accessible que par ces opérations, et la manière dont il est implémentée reste cachée à l'utilisateur. Sur ce principe, lorsque l'on utilise une voiture, la manière dont est fabriquée la carburation importe peu. Seule les opérations disponibles (démarrer, rouler, stopper, ...) sont utilisables pour le conducteur.

L'implémentation cachée d'un objet est réalisée par ses attributs. Les opérations utilisables sont définies par les méthodes. On peut donc voir les méthodes comme l'implémentation des services que vont rendre toutes les instances d'une classe.

Puisqu'il n'est pas souhaitable d'accéder directement aux attributs, il est nécessaire d'implémenter des méthodes pour accéder aux valeurs de ces attributs et pour les modifier. On appelle ces méthodes des **accesseurs** : les accesseurs en lecture (consultation) et les accesseurs en écriture (modification).

Nous prendrons comme habitude de précéder le nom des accesseurs en lecture par *get* et les accesseurs en écriture par *set*.

Dans l'exemple suivant, l'attribut *valeur* est accompagné des deux accesseurs $getValeur$ et $setValeur$:

```
class Test {
    int valeur;

    int getValeur(){
       return this.valeur;
    }
    void setValeur(int v){
       this.valeur = v;
    }
}
```

2.4.4 La méthode *main*

La méthode *main* est le point d'entrée d'un programme. Ce point d'entrée doit être unique car sinon il serait impossible de savoir où commencer le programme.

La méthode *main* ne renvoie aucune valeur (contrairement au langage *C*) et prend en paramètre un tableau de chaîne de caractères. Ce tableau contient éventuellement les paramètres qui sont passés sur la ligne de commande lors de l'appel de l'application.

En Java, la méthode *main* est déclarée *statique*. Cela signifie que cette méthode est une méthode de classe et non d'instance.

Les méthodes que nous avons décrites jusqu'à présent sont dites des méthodes d'instance, car elles s'appliquent uniquement sur une instance d'une classe. Il existe également un autre type de méthode dite *méthode de classe*.

Les méthodes de classe sont indépendantes de toutes les instances et ne s'appliquent pas aux instances mais à la classe directement. En Java, les méthodes statiques sont précédées du mot clef *static*.

3 Principe de la récursivité

3.1 Définition

Un sous programme est récursif lorsqu'il s'appelle lui même. Par exemple on pourrait définir la fonction *factorielle* de manière suivante :

```
int factorielle(int n){
   if(n==0) return 1;
   else return n * factorielle(n-1);
}
```

Cette définition récursive de la factorielle provient directement de la définition mathématique suivante :

$$\begin{cases} 0\,! = 1 \\ n\,! = n * (n-1)! \end{cases}$$

L'intérêt de la récursivité est que le raisonnement sous jacent est plus proche du raisonnement mathématique et par conséquent plus formel, contrairement au raisonnement itératif qui reste plus proche du fonctionnement des ordinateurs. D'autre part, on peut considérer que le fonctionnement récursif est proche du raisonnement humain.

Voici une version itérative de la fonction factorielle :

```
int fact(int n) {
      int result=1, i ;
      for(i=1; i<=n; i++)
          result = result*i;
      return result;
}
```

Dans cette version itérative, on voit l'apparition d'une variable *result* qui permet de faire varier au fur et à mesure le calcul temporaire de la factorielle. On peut remarquer qu'aucune variable n'est nécessaire dans la version récursive ; le calcul est en fait géré au niveau des retours successifs des appels de fonction : on parle de *pile d'exécution* (voir 3.2). L'initialisation à 1 de la variable *result* correspond au cas où n vaut 0 dans la version récursive.

La plupart des algorithmes peuvent s'écrire de manière récursive et de manière itérative. Certains algorithmes sont naturellement itératifs, alors que d'autres sont plus naturellement récursifs. C'est ce critère qui permet principalement de choisir entre les deux implémentations.

Il faut cependant noter que certains algorithmes comportant plus de deux appels récursifs sont difficiles voire impossibles à écrire facilement de manière itérative.

3.2 Fonctionnement de la récursivité

Exemple d'utilisation On identifie quatre types de récursivités : la récursivité simple, la récursité multiple, la récursivité mutuelle et la récursivité imbriquée.

Une fonction est *simplement récursive* lorqu'elle ne s'appelle récursivement qu'une seule fois. Par exemple, la fonction *puissance* qui a x associe x^n, possède une récursivité simple. Elle peut être définie comme suit :

$$x^n = \begin{cases} 1 & si\ n = 0 \\ x * x^{n-1} & si\ n \leq 1 \end{cases}$$

Sa traduction en langage *Java* serait :

```
int puissance(int x, int n){
    if(n==0) return 1;
    else     return x * puissance(x,n-1);
}
```

Une définition d'une fonction *récursive multiple* peut contenir plus d'un appel récursif. La suite de Fibonacci possède par exemple deux appels récursifs et se définit comme suit :

$$\begin{cases} u_1 = 1 \\ u_2 = 1 \\ u_{n+2} = u_{n+1} + u_n \quad pour\ n > 2 \end{cases}$$

Sa traduction serait :

```
int fibonacci(int n){
    if(n==1 || n ==2) return 1;
    else return fibonacci(n-1) * fibonacci(n-2);
}
```

Des définitions sont dites *mutuellement récursives* lorsqu'elles dépendent les unes des autres. C'est le cas pour la définition suivante de la parité :

$$pair(n) = \begin{cases} vrai & si\ n = 0 \\ impair(n-1) & sinon \end{cases} \qquad impair(n) = \begin{cases} faux & si\ n = 0 \\ pair(n-1) & sinon \end{cases}$$

La fonction *pair* appelle bien la fonction *impair*, et inversement. Leur code en langage *Java* pourrait être :

```
int pair(int n){
     if(n==0) return 1;
     else     return impair(n-1);
}
int impair(int n){
     if(n==0) return 0;
     else     return pair(n-1);
}
```

Enfin, il existe des définitions qui nécessitent des appels de récursion imbriqués. C'est le cas dans la définition de la fonction d'Ackermann définie comme suit :

$$A(m,n) = \begin{cases} n+1 & si\ m = 0 \\ A(m-1,1) & si\ m > 0\ et\ n = 0 \\ A(m-1, A(m,n-1)) & sinon \end{cases}$$

On peut effectivement voir dans le dernier cas que l'appel de la fonction contient en paramètre un second appel récursif. Cette fonction ne peut pas par ailleurs s'écrire par une récursion simple. Voici un exemple d'implémentation de cette fonction :

```
int ack(int m, int n){
     if(m==0) return n+1;
     else if(m>0 && n==0) return ack(m-1,1)
     else ack(m-1, ack(m, n-1));
}
```

Mode d'emploi écrire un programme de manière récursif nécessite d'avoir, comme un problème récurrent en mathématiques :

- un certain nombre de cas dont la résolution est connue, cas que l'on dénommera *cas particulier*, et qui peuvent être considérés comme des points d'arrêt de la récursion,
- un moyen de ramener le *cas général* à un cas plus simple.

Pour qu'un appel à un algorithme récursif soit correct, il faut s'assurer qu'il retombera toujours sur un cas particulier. En quelque sorte, soit un problème est assez simple pour que l'on puisse y répondre directement, soit il se **décompose en sous-problèmes** et **combine ensuite les résultats** de ces sous problèmes pour former la solution.

On retrouve ce concept général dans l'algorithme du tri par fusion (voir **??**) dont voici le code :

```
void tri_fusion(int [] tableau, int deb, int fin) {
   // cas d'arrét deb=fin: on ne fait rien
   if (deb!=fin) {
       int milieu=(fin+deb)/2;
       // étape de séparation:
       tri_fusion_2(tableau,deb,milieu);
       tri_fusion_2(tableau,milieu+1,fin);
       // étape de fusion:
       fusion(tableau,deb,milieu,fin); }
   }
}
```

Typiquement, la phase de décomposition en sous problèmes doit converger vers un des cas particuliers. Si ce n'est pas le cas, l'algorithme ne s'arrêtera pas.

Notion de Pile d'exécution Revenons sur l'exemple de la fonction factorielle :

```
1   int factorielle(int n){
2       if(n==0) return 1;
3       else {
4          int i = factorielle(n-1);
5          return n * i;
6   }
```

La ligne 2 correspond au cas particulier qui assure l'arrêt de la récursion. La ligne 4 correspond au cas général qui réduit le problème à l'appel de $factorielle(n-1)$. La phase de combinaison se trouve à la ligne 5, où l'on renvoie la multiplication du résultat de l'appel récursif par n.

La figure 1 présente le déroulement de l'appel de $factorielle(4)$. $factorielle(4)$ appelle successivement $factorielle(3)$, $factorielle(2)$, $factorielle(1)$ puis $factorielle(0)$. Ce dernier cas correspond au cas d'arrét de la récursion ; la valeur 1 est renvoyée. Comme l'on connaét la valeur de la $factorielle(0)$, on peut déterminer la valeur de $factorielle(1)$, *etc*, jusqu'à le calcul de $factorielle(4)$. Comme vu précédemment, on retrouve bien les deux phases de décomposition et de combinaison : les différents appels récursifs successifs correspondent à la phase de décomposition en un sous problème, et les différentes valeurs calculées et retournées correspondent bien à la phase de combinaison des résultats.

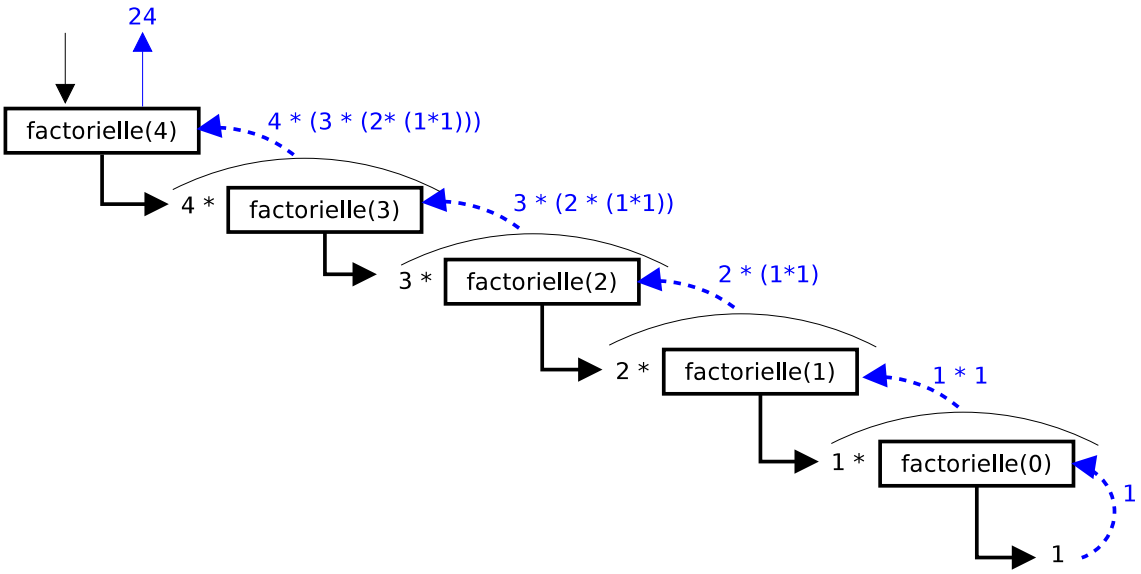


FIGURE 1 – Exemple de déroulement de l'appel récursif $factorielle(4)$

On peut voir dans la figure 1 que l'ordinateur a besoin de mémoriser les différents appels successifs de la fonction. En particulier, les différents paramètres, les variables locales d'une fonction ainsi que les adresses de retour des fonctions sont mémorisés dans la **pile d'exécution** du programme.

Comme son nom l'indique, la pile d'exécution repose sur le principe LIFO (dernier arrivé, premier sorti). Il ne faut pas perdre en ligne de compte que la taille de cette pile d'exécution possède une taille fixe qui ne peut varier en cours d'exécution. Un trop grand nombre d'appels récursifs ou une taille excessive des paramètres, entraînerait un **débordement de pile**, et donc l'arrêt du programme.

4 Les mots réservés du langage Java

abstract Il s'agit d'un modificateur utilisé lors de la déclaration d'une *classe* ou d'une *méthode*. On dit alors que la classe ou la méthode est abstraite.

Une méthode abstraite n'est composée que de son prototype, c'est-à-dire son type de retour suivi de son nom, de la liste des paramètres entre parenthéses, et d'un point-virgule. Une méthode abstraite ne peut pas être déclarée *static*, *private* ou *final*.

Dès qu'une classe contient une méthode abstraite, elle doit elle aussi être déclarée abstraite. Une classe abstraite ne peut pas être instanciée. Il faudra l'étendre c'est-à-dire faire une sous classe, et définir toutes les méthodes abstraites que cette classe abstraite contient.

```
abstract class Forme {
    public abstract int aire();
    //...
}

class Rectangle extends Forme  {
    int hauteur, largeur;
    //...
    public int aire(){
        return hauteur * largeur;
    }
}
```

Dans l'exemple précédent, la classe *Rectangle* hérite de *Forme*. Comme à toute forme, on peut définir son aire, la méthode *aire*() est déclarée dans la classe *Forme*. Par contre, aucune formule générale n'existe pour donner l'aire de toutes les formes ; aussi la méthode *aire*() est donc déclarée abstraite dans *Forme*. Par contre, l'aire d'un rectangle pouvant se calculer, la fonction est réécrite dans *Rectangle*.

assert (depuis la version 1.4) Il s'agit d'un mécanisme qui permet de tester des hypothèses dans le programme. C'est ce que l'on appelle la programmation par contrat. Le but est de limiter le nombre d'erreurs dans les programmes.

Dans l'exemple suivant, la méthode *setWidth* vérifie que la largeur d'un rectangle est bien supérieure à 0 avant d'affecter la variable *largeur*.

```
class Rectangle {
    int hauteur, largeur;
    ...
    public void setWidth(int s) {
        assert(s>0);
        largeur = s;
    }
}
```

boolean Type possédant les deux seules valeurs suivantes : *true* (vrai) et *false* (faux).

break L'instruction *break* force la sortie d'un bloc, d'un choix (*switch*) ou d'une boucle (*for*, *while* ou *do while*). Mais outre dans les instructions *switch*, son utilisation ne fait pas partie des bonnes pratiques de programmation en Java.

byte Type primitif qui désigne des valeurs entières signées sur 8 bits, comprise entre -128 et $+127$ inclus.

char Type primitif qui désigne les caractères *unicode* codé sur $16bits$. Les valeurs vont de $\backslash u0000$ à $\backslash ufffff$.

case Voir l'instruction *switch*.

catch Voir l'instruction *try*.

class Cette instruction permet de définir une classe.

```
class Rectangle {
    ...
}
```

const Non utilisé.

continue L'instruction *continue* n'a de signification que dans une itération de boucles (*for*, *while* ou *do while*). Les instructions jusqu'à la fin de la boucle sont passées ; l'exécution de la boucle reprend ensuite.

```
int i;
...
while( i<10 ) {
   if(i == 5) {
      continue;
   }
   bloc_d_instructions
}
```

L'utilisation de *continue* n'est pas courante et ne fait pas partie des bonnes pratiques de programmation. Dans l'exemple précédent, lorsque i vaut 5, les bloc d'instructions n'est pas exécuté. La boucle *continue* pour une autre itération.

default Voir l'instruction *switch*.

do while La syntaxe de l'instruction *do while* est la suivante :

```
do {
      instruction(s)
} while (expression booléenne) ;
```

Une fois que l'instruction ou les instructions sont effectuées, l'expression est évaluée. Tant que l'expression est vérifiée, l'instruction ou les instructions sont effectuées de nouveau.

À la différence de *while*, l'instruction ou les instructions sont au moins faites une fois, puisque l'expression est évaluée à la fin de l'itération.

double type primitif qui désigne des valeurs décimales en double précision sur 64 bits.

else Voir instruction *if*.

enum (depuis la version 5.0) Un type énuméré correspond à un ensemble de valeurs constantes et prédéfinies. Par exemple, soit la déclaration suivante du type *Semaine* :

```
enum Semaine {
      LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE
}
```

La déclaration et l'affectation des variables de type *enum* s'effectuent de la manière suivante :

```
Semaine s = Semaine.LUNDI;
```

Le type *enum* peut aussi spécifier des valeurs pour les constantes, et contenir des méthodes qui lui sont propres (*cf* javadoc).

extends Ce mot clef spécifie un lien d'héritage entre deux classes ou deux interfaces. Dans l'exemple suivant, la classe *Carre* hérite de la classe *Rectangle*. La raison en est que l'on peut considérer qu'un carré est un rectangle particulier dont la largeur et la hauteur sont égales.

```
class Rectangle {
    ...
}
class Carre extends Rectangle {
    ...
}
```

false Voir le type *boolean*.

final Il s'agit d'un modificateur utilisé lors de la déclaration d'une *classe*, d'une *méthode* ou d'une *variable*.

Une variable ou une méthode finale indique que la variable ou la méthode ne pourra pas être redéfinie dans une sous classe. Une classe finale ne pourra pas avoir de sous classe (comme par exemple la classe *String*).

finally Voir l'instruction *try*.

float type primitif qui désigne des valeurs décimales en simple précision sur 32 bits.

for La boucle *pour* possède deux syntaxes.

La première est issue du langage C. Elle permet de répéter une certain nombre de fois une suite d'instructions. La syntaxe est la suivante :

```
for( initialisation ; terminaison ; incrément ) {
      instruction(s)
}
```

La partie *initialisation* n'est effectuée qu'une seule fois au début de la boucle. Elle peut contenir plusieurs instructions séparées chacune par une virgule.

La partie *terminaison* correspond à une expression booléenne.

La partie *incrément* sera effectué en fin de boucle, une fois que le bloc d'instructions sera effectué. Cette partie peut également contenir plusieurs instructions, séparées chacune par une virgule.

La deuxiéme syntaxe a été introduite à partir de la version 5 de java, et concerne uniquement les éléments qui implémentent l'interface *Iterable*. C'est le cas de la classe *ArrayList* comme dans l'exemple suivant :

```
ArrayList<String> maListe = new ArrayList<String>();
...
for(String s : maListe) {
   System.out.println(s);
}
```

goto (non utilisé)

if La syntaxe de l'instruction *if* est la suivante :

```
if (expression_booléenne) {
    bloc_d_instructions_alors
} else {
    bloc_d_instructions_sinon
}
```

En fonction de la valeur de l'expression booléenne, l'instruction if détermine quel bloc d'instructions sera exécuté. Si l'expression est vérifiée, le bloc d'instructions *alors* sera exécuté ; dans le cas contraire, c'est le bloc d'instructions *sinon* qui le sera.

La partie *sinon* n'est pas obligatoire.

Les accolades sont optionnelles dans le cas où le bloc d'instructions n'est composé que d'une seule instruction.

Les parties *alors* et *sinon* peuvent contenir d'autres instructions if. On peut alors obtenir des instructions conditionnelles en cascade. Dans l'exemple suivant, en fonction de la valeur de la variable *age*, on précise la valeur de la variable *tarif* [6] :

```
int age, tarif;
...
if(age<4)
    tarif = 0;
else if(age >=4 && age <12)
    tarif = 5;
else if(age>=12 && age <18)
    tarif = 10;
else tarif = 12;
```

La partie *sinon* étant optionnelle, il se peut qu'il y ait ambiguité lorsque deux instructions if sont imbriquées, et qu'il n'y ait qu'une seule partie *sinon*. Par défaut, la partie *sinon* sera rattachée au if le plus proche, comme il est montré dans l'exemple suivant :

```
if(a==1)
   if(b==2)    // partie 'sinon' rattachée à ce 'if'
      c = 3;
   else        // partie 'sinon'
      c = 4;
```

implements Ce mot-clef est utilisé pour spécifier quelle *interface* doit être implémentée dans une classe. Dans l'exemple suivant, la classe A doit implémenter l'interface B :

```
class A implements B {
    ...
}
```

import Le mot clef *import* spécifie dans quel paquetage on peut trouver certaines classes qui ne sont pas dans le paquetage de la classe. On peut voir son utilisation comme un moyen de ne pas écrire le nom complet des classes. La syntaxe est la suivante :

```
import nom_de_paquetage.nom_de_classe;
```

Ainsi, si l'on veut déclarer une instance de la classe *Random*, on pourra écrire :

```
import java.util.Random;
...
Random maVariable = new Random();
```

Sans l'*import*, on devrait écrire :

6. Dans cet exemple, on considère que la variable *age* est nécessairement supérieure à 0.

```
java.util.Random maVariable = new java.util.Random();
```

On peut aussi importer l'ensemble des classes d'un paquetage de la manière suivante :

```
import java.util.*;
```

On peut ainsi utiliser toutes les classes du paquetage *java.util*.

Il est également possible d'importer des méthodes ou champs statiques d'une classe, de telle sorte que l'on puisse les utiliser comme s'ils faisaient partie de la classe. Ainsi, au lieu d'écrire le code suivant :

```
double d = Math.cos(1);
```

on pourra avoir :

```
import java.lang.Math.cos;
...
double d = cos(1);
```

On peut importer autant de paquetages que l'on veut. Toutefois, si deux paquetages importés possèdent chacun une classe du même nom, les classes doivent être spécifiées par leur nom complet afin d'enlever toute ambiguïté.

instanceof L'opérateur *instanceof* permet de savoir si une variable est bien une instance d'une classe particuliére. Cela prend en compte les liens d'héritage entre les classes.

Soit par exemple les classes *Animal* et *Oiseau* où *Oiseau* hérite de *Animal* :

```
class Animal { ... }
class Oiseau extends Animal { ... }
```

Par héritage, tout oiseau est un animal, ce qui implique bien que la variable *o* ci-dessous est à la fois une instance de la classe *Oiseau* mais aussi de la classe *Animal*.

```
Oiseau o = new Oiseau();
o  instanceof Animal   // vrai
o  instanceof Oiseau   // vrai
```

La variable $a1$ ci-dessous est juste affectée quant à elle à une instance de la classe *Animal*, et n'est donc pas logiquement une instance de la classe *Oiseau*.

```
Animal a1 = new Animal();
a1 instanceof Animal   // vrai
a1 instanceof Oiseau   // faux
```

Comme un oiseau est un animal particulier (lien d'héritage), il est possible d'affecter à une variable *Animal* une instance de la classe *Oiseau*. Ainsi, $a2$ est une instance des classes *Animal* et *Oiseau*.

```
Animal a2 = new Oiseau();
a2 instanceof Animal   // vrai
a2 instanceof Oiseau   // vrai
```

int Type primitif qui désigne des valeurs entières signées sur 32 bits.

interface Permet de définir une *comportement* ou une *caractéristique* qui est le/la même pour un certain nombre de classes.

Une interface est composée de méthodes où le code n'est pas spécifié. Une classe qui se veut avoir le comportement d'une interface devra *implémenter* l'interface, et devra écrire le code de toutes ces méthodes. Si elles ne sont pas écrites, la classe sera considérée comme abstraite.

Il est possible de définir également dans une interface des constantes (*public static final*).

long Type primitif qui désigne des valeurs entières signées sur 64 bits.

native Ce mot clef est utilisé pour indiquer qu'une méthode n'est pas écrite en Java mais dans un autre langage natif (en *C* ou *C++* par exemple). Une fonction native en Java n'aura par conséquent pas de corps.

```
public native void fonctionExterieure();
```

Cette méthode devra nécessairement être écrite dans un fichier séparé ; le lien est réalisé par l'intermédiaire de bibliothéque partagée. Ce lien côté Java se fait à l'aide de la méthode statique *System.loadLibrary*(*String name*). La bibiothéque devra nécessairement se situer dans l'un des chemins listés par la propriété *java.library.path*.

new Cet opérateur permet d'associer à une variable une nouvelle instance de la classe considérée.

Cette création d'instance se déroule en deux phases : la première est une réservation de la place mémoire nécessaire pour stocker cette instance, la seconde correspond à un appel à l'un des constructeurs de la classe. Le constructeur est appelé en fonction des paramètres passés en paramètre de l'opérateur *new*.

```
String monMot;    // déclaration d'une variable de type String
monMot = new String("bonjour"); // création d'une instance de type String
            // c'est le constructeur String(char [] value) qui est appelé
int lg = monMot.length();       // renvoie la longueur de l'instance
```

null *null* est une valeur spéciale indiquant qu'une variable n'est associée à aucune instance. On parle de *référence nulle*. *null* est la valeur par défaut de toute variable (autre que les variables de type primitif).

```
String monMot = null;
```

Si un appel à une méthode est réalisé sur une variable dont la valeur est *null*, comme dans l'exemple suivant, l'exception *NullPointeurException* est levée :

```
String monMot = null;
int lg = monMot.length(); // l'exception NullPointerException est levée
```

package Java permet de structurer les classes en les regroupant dans un même paquetage, par exemple en fonction de leur thématique. L'instruction *package* sera nécessairement la première ligne de code du fichier et sera suivie du nom du paquetage.

```
package monPaquetage;
```

Si aucun paquetage n'est spécifié, la classe appartiendra alors au paquetage par défaut (ce paquetage est sans nom) ; ce paquetage sera composé de toutes les classes qui se trouvent dans le même répertoire et au même niveau.

Une classe aura accès aux autres classes du même paquetage. Lorsque le programme aura besoin d'une classe se trouvant dans un autre paquetage, il est nécessaire d'indiquer en plus du nom de la classe, le nom du paquetage auquel elle appartient. On peut également pour simplifier l'écriture du code utilisé l'instruction *import*.

private *private* est un modificateur de visibilité qui déterminent le degré d'accessibilité des champs ou des méthodes. Un champ ou une variable privé n'est visible que depuis sa propre classe. Aucune autre classe n'y a accès.

```
class MaClasse {
   private int maVariable;
   ...
}
```

protected c'est un modificateur de visibilité qui déterminent le degré d'accessibilité des champs ou des méthodes. Un champ ou une méthode classe protégé n'est visible que depuis le paquetage dans lequel il est défini, ainsi que dans les sous classes de la classe où il est défini.

```
class MaClasse {
   protected int maVariable;
   ...
}
```

public il s'agit d'un modificateur de visibilité qui déterminent le degré d'accessibilité des classes, des champs ou des méthodes.

Lorsqu'une classe est déclarée publique, cela signifie que cette classe est visible depuis l'ensemble du code, y compris depuis les autres paquetages. Si la classe n'est pas spécifiée publique, alors la classe n'est visible que depuis le paquetage dans lequel elle est définie.

```
public class MaClasse {
   ...
}
```

Une variable ou une méthode classe publique est visible depuis n'importe où dans le code.

```
public class MaClasse {
   public int maVariable;
   ...
}
```

return Cette instruction permet de stopper l'exécution d'une méthode, et de renvoyer éventuellement une valeur. L'instruction *return* sera la dernière instruction exécutée dans la méthode.

```
int addition(int a, int b){
     return a+b;
}
```

Il peut aussi y avoir plusieurs *return* dans une méthode comme dans l'exemple suivant :

```
int factorielle(int a){ // renvoie la factorielle de a
     if(a==0) {
        return 1;
     } else {
        return n * factorielle(n-1);
     }
```

short Type primitif qui désigne des valeurs entières signées sur 16 bits.

static Ce mot clef indique qu'une variable, une méthode ou un bloc d'instruction n'est pas associé à une instance particulière d'une classe. C'est le cas notamment de la méthode *abs* du paquetage *java.lang.Math*, méthode qui renvoie la valeur absolue d'une valeur :

```
double d = Math.abs(-1);
```

Toutes les instances partageront les variables ou méthodes statiques.

Une méthode statique n'étant liée à aucune instance, elle ne pourra accéder qu'à des variables ou méthodes elles-mêmes statiques.

Les blocs statiques sont utilisés pour initialiser des variables statiques. Ils sont exécutés à l'initialisation de la classe.

```
public class MaClasse {
  ...
```

```
  public static ArrayList list = new ArrayList();
  static {
    list.add(new String("a");
    list.add(new String("b");
  }
}
```

strictfp (depuis la version 1.2) Ce mot clef peut être utilisé dans la déclaration des classes, interface ou méthodes. *strictfp* est une abréviation de *strict floating point*.

Cela garantit la précision et les arrondis effectués avec les flottants pour assurer la portabilité entre les différentes machines virtuelles Java selon la norme $IEEE$ 754.

super *super* désigne la sur-classe d'une classe. Deux utilisations sont possibles.

La première utilisation de *super* est lié à l'accès aux variables et méthodes de la sur-classe. Il se peut en effet, que l'on veuille modifier la valeur d'une variable ou le code d'une méthode d'une sous classe. On peut tout de même vouloir accéder à une variable ou une méthode de la sur-classe en utilisant le mot clef *super*. Soit l'exemple suivant où la classe *Oiseau* hérite de la classe *Animal*.

```
class Animal {
    String getName(){
        return "animal";
    }
}
class Oiseau extends Animal {
    String getName(){
        return "oiseau-" + super.getName();
    }
}
```

Dans l'exemple ci-dessus, la méthode $getName$ de la classe *Animal* renverra la chaîne "$animal$" tandis que la méthode $getName$ de la classe *Oiseau* renverra "$oiseau - animal$".

La deuxième utilisation du *super* permet de faire appel dans un constructeur au constructeur de la sur-classe, en mettant entre parenthèses les paramètres nécessaires pour ce dernier. Dans l'exemple suivant, la classe *Carre* hérite de la classe *Rectangle*, dans la mesure où un carré est un rectangle particulier dont la hauteur et la largeur sont identiques ; le constructeur de la classe *Carre* fait appel au constructeur de la classe *Rectangle* :

```
class Rectangle {
    int largeur;
    int hauteur;
    public Rectangle(int l, int h){
        largeur = l;
        hauteur = h;
    }
}
class Carre extends Rectangle {
    public Carre(int cote){
        super(cote,cote);
    }
}
```

switch La structure de contrôle *switch* permet d'exécuter des instructions en fonction d'une valeur. Elle remplace avantageusement de nombreuses structures conditionnelles if imbriquées.

```
int a;
...
switch(a) {
   case 1:  // instructions pour a=1
            break;
   case 2:  // instructions pour a=2
            break;
   default: // instructions pour les autres valeurs de a
}
// instructions après le switch
```

Le mot clef *case* n'est pas une instruction, mais une étiquette. Il s'agit plus en fait d'un repére dans le code. En fonction de la valeur de a, le programme va directement au *case* concerné. Toutes les valeurs traitées par le *switch* doivent être par conséquent différentes.

L'utilisation de l'instruction *break* est indispensable, car elle permet d'aller directement aux instructions après le *switch*. Si un *break* est omis, les instructions du *case* suivant seront également exécutées.

Le cas par défaut ($default$) est nécessaire dans la mesure où toutes les valeurs de a doivent être traitées impérativement par le *switch*.

synchronized Ce mot clef est utilisé dans le cadre de la programmation concurrente (multi-processus). Il caractérise un bloc d'instructions ou une méthode qui ne pourra pas être exécuté en même temps par deux threads. Il se peut en effet qu'un même code soit potentiellement exécuté par deux threads ; or certaines ressources peuvent être critiques et leur accès doit être régulé pour éviter que leur valeur soit incertaine.

```
int i;
...
public void synchronized fonctionEnRessourceCritiques(int i){
  maValeur = i;
}
```

this *this* référence la classe courante. Deux utilisations sont possibles.

La première utilisation est liée à l'accès aux variables et méthodes de la classe courante. L'utilisation de *this* est parfois indispensable dans le cas où une variable locale porte le même nom qu'une variable de la classe. Dans les autres cas, l'utilisation de *this* est superflue.

```
class Carre {
    int cote; // variable globale à la classe
    public Carre(int cote){ // cote: variable locale à la méthode
        this.cote = cote;
    }
}
```

Dans l'instruction $this.cote = cote$, le $this.cote$ désigne la variable globale à la classe, alors que $cote$ désigne la variable locale à la méthode.

La deuxième utilisation de *this* permet de faire appel à un autre constructeur de la classe. Ce sont les paramètres passé à l'appel qui détermine lequel des constructeurs est appelé. Dans l'exemple suivant, le constructeur par défaut (constructeur sans paramètre) appelle le second constructeur avec h et l qui valent 10 :

```
class Rectangle {
    int largeur;
```

```
    int hauteur;
    public Rectangle(){
        this(10,10);
    }
    public Rectangle(int l, int h){
        largeur = l;
        hauteur = h;
    }
}
```

throw Cette instruction permet de lancer une exception. Lorsqu'une exception est lancée, l'exécution de la méthode dans laquelle elle se trouve, s'arrête.

```
...
if(a==1)
    throw new Exception("erreur détectée");
...
```

throws Ce mot clef est utilisé dans la déclaration des méthodes qui peuvent lever une exception. Son utilisation est obligatoire.

```
void fonction() throws Exception {
  ...
  if(a==1)
     throw new Exception("erreur détectée");
  ...
}
```

transient Ce mot clef est utilisé pour indiquer qu'une variable globale d'une classe ne sera pas sauvée lors de la sérialisation d'une instance de la classe.

```
class Test implements Serializable {
  transient int variableNonSerialisee;
  int variableSerialisee;
  ...
}
```

true Voir le type *boolean*.

try Cette instruction permet d'exécuter un code dans lequel une exception pourait être lancée. C'est l'instruction *catch* qui permet de traiter l'exception levée.

Plusieurs *catch* peuvent se succéder.

L'instruction *finally* est optionnelle et contiendra un code qui sera exécuté quoi qu'il arrive après les blocs d'instructions du *try* ou des *catch*, et ceci même si une instruction *return* est rencontrée dans l'un de ces blocs.

```
void fonction() throws Exception, AutreException {
   ...
}
...
void fonction2() {
    try {
       fonction();
    } catch (Exception e1) {
       // code pour traiter l'exception
```

```
        } catch (AutreException e2) {
            // code pour traiter la seconde exception
        } finally {
            // partie du code qui sera dans tous les cas exécutée
            // après les blocs d'instructions du try ou du catch
        }
    }
```

void Le mot clef *void* stipule qu'une méthode ne renverra aucune valeur.

```
    void affiche(){
        System.out.println("bonjour");
    }
```

volatile Ce mot clef est utilisé dans le cadre de la programmation concurrente (multi-processus). Il garantit, lors d'un accès à une variable globale d'une classe, que la valeur soit bien la valeur la plus récente, et non pas une valeur qui aurait été copiée en cache.

```
volatile int a = 1;
```

while Cette instruction permet d'exécuter un bloc d'instruction tant qu'une condition est vérifiée. La condition est vérifiée à chaque début de l'itération.

```
while( condition ) {
    // instructions
}
```

Table des matières