INITIATION A JAVA



**Patrice E. BROU**

2012

Unité :1

Thème : **Java 2**, les concepts de base

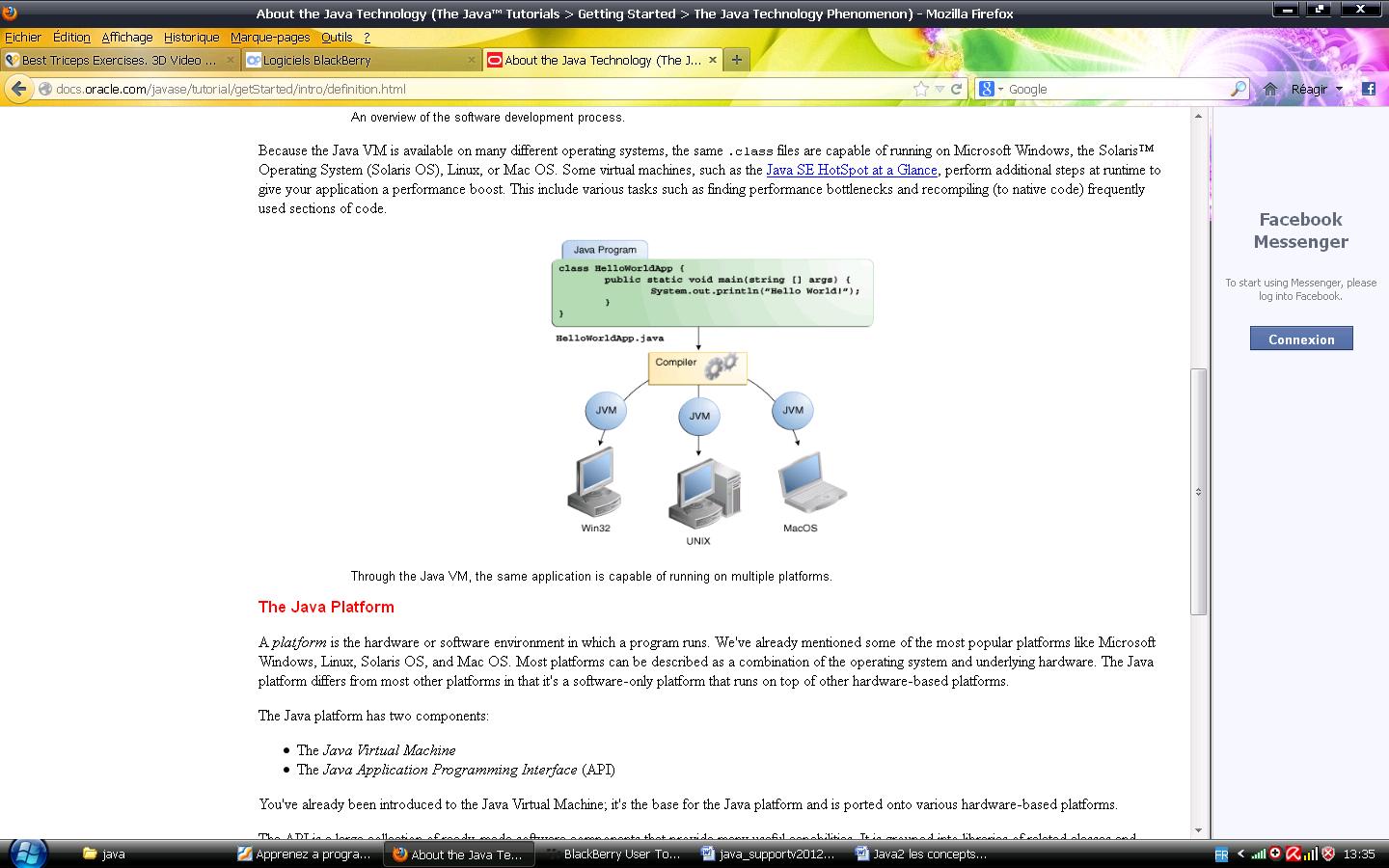
**Historique**

Apparu fin 1995 début 1996 et développé par Sun Microsystems Java s'est très rapidement   
taillé une place importante en particulier dans le domaine de l'internet.

Destiné au départ à la programmation de centraux téléphoniques sous la désignation de langage "oak", la société Sun a eu l'idée de le recentrer sur les applications de l'Internet et des réseaux. C'est un langage en évolution permanente Java 2 est la version stabilisée de java fondée sur la version initiale 1.2.2 du **JDK** (Java Development Kit) dont la dernière version est Java SE 1.7 update 9.

NB : Les éditeurs de logiciels dont Oracle recommandent la version 1.6 qui est plus stable.

Les objectifs de java sont d'être multi plateformes et d'assurer la sécurité aussi bien pendant le développement que pendant l'utilisation d'un programme java.). C’est un langage algorithmique et orientée objet comme C#et C++. Son utilité n’est plus à démontrer dans les domaines bureautiques, graphiques, multimédias, bases de données, environnement de développement, etc...)



Les technologies java offrent :

* Des outils de développement permettant de créer compiler et exécuter et documenter des programmes ;
* Des API ( application programme interface) mettant en œuvre les fonctionnalités du langage JAVA depuis la création d’objets de base jusqu’ a la programmation réseau ;
* Des technologies de déploiement dans différentes plate formes architecturales de logiciel ;
* Des bibilothéques graphiques permettant de réaliser des IHM tres evolués ;
* Des librairies integrées pour uin accès aux bases de données.

11 - Les éléments du langage

11.2 - Les types de base  en Java

Java est un langage fortement typé : Comme les langages C, C++, Pascal etc. Java dispose de type prédéfinis ou type élémentaires. En Java, Il faut déclarer un objet ou une variable avec son type avant de l'utiliser.

*2.1 Tous les type élémentaires de Java et le transtypage*

|  |  |  |
| --- | --- | --- |
| ***type élémentaire*** | ***intervalle de variation*** | ***nombre de bits*** |
| **boolean** | false , true | 1 bit |
| **byte** | -128 ... +127 | 8 bits |
| **char** | caractères unicode (valeurs de 0 à 65536) | 16 bits |
| **double** | Virgule flottante double précision ~5.10^308 | 64 bits |
| **float** | Virgule flottante simple précision ~9.10^18 | 32 bits |
| **int** | entier signé -2^31-1... +2^31 | 32 bits |
| **long** | entier signé long -2^63-1... +2^63 | 64 bits |
| **short** | entier signé court -2^15-1... +2^15 | 16 bits |

***Remarque : Il est possible d'indiquer au compilateur le type d'une valeur numérique en utilisant un suffixe***  
**l** ou **L** pour désigner un entier du type long

**f** ou **F** pour désigner un réel du type float :

**d** ou **D** pour désigner un réel du type double.

***Exemples :***

45l  ou 45L représente la valeur 45 en entier signé sur 64 bits.

45f  ou 45F représente la valeur 45 en virgule flottante simple précision sur 32 bits.

45d  ou 45D représente la valeur 45 en virgule flottante double précision sur 64 bits.   
5.27e-2f  ou 5.27e-2F représente la valeur 0.0527 en virgule flottante simple précision sur 32 bits.

***Transtypage opérateur (  )***

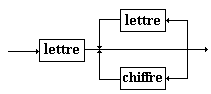
(**pas de conversion implicite**). Pour convertir une valeur immédiate ou une valeur contenue dans une variable il faut explicitement transtyper cette valeur à l'aide de l'opérateur de transtypage noté: **( )**.

**int x ;   
x = (int) y ;**

*signifie que l’on demande de transtyper la valeur contenue dans la variable y en un entier signé 32 bits avant de la mettre dans la variable y.*

* Tous les types élémentaires peuvent être transtypés à l'exception du type **boolean** qui ne peut pas être converti en un autre type (différence avec le C).
* Les conversions **peuvent être restrictives** quant au résultat; par exemple le transtypage du réel 5.27e-2 en entier ( x = **(int)**5.27e-2)  mettra l'entier zéro dans x.

*2.2 Variables, valeurs, constantes en Java*   
 **Une variable** Java peut contenir soit une valeur d'un type élémentaire, soit une référence à un objet. Les variables jouent le même rôle que dans les langages de programmation classiques impératifs. La règle d’écriture d’un identificateur est donnée par le schéma ci-dessous.

Identificateur Java :   
lettre Є {a…Z} et {0…9}  
Attention Java est casse sensitif, c'est à dire que la variable **BonJour** n'est pas la même que la variable **bonjour** ou encore la variable **Bonjour**. En plus des lettres, les caractères suivants sont autorisés pour construire une identificateur Java : "$" , "\_" , "µ" et les lettres accentuées.

Exemples de déclaration de variables :

**int** Bonjour ;  **int** µEnumération\_fin$;   
**float** Valeur ;   
**char** UnCar ;   
**boolean** Test ;

etc ...

Exemples d'affectation de valeurs à ces variables :

|  |  |
| --- | --- |
| ***Affectation*** | ***Déclaration avec initialisation*** |
| Bonjour = 2587 ;  Valeur = -123.5687  UnCar =  'K' ;  Test = false ; | **int** Bonjour = 2587 ;  **float** Valeur = -123.5687  **char** UnCar =  'K' ;  **boolean** Test = false ; |

***Exemple avec transtypage :***   
**int** Valeur ;   
**char** car = '8' ;   
Valeur = (**int**)car - (**int**)'0';

Lorsque la variable car est l'un des caractères '0', '1', ... ,'9', la variable Valeur est égale à la valeur numérique associée (il s'agit d'une conversion car = '0' ---> Valeur = 0, car = '1' ---> Valeur = 1, ... , car = '9' ---> Valeur = 9).

Les constantes en Java ressemblent à celles du pascal

Ce sont des variables dont le contenu ne peut pas être modifié, elles sont précédées du mot clef **final** :

*Exemple*

**final** **int** x=10 ; *x est déclarée comme constante entière initialisée à 10.*   
  x = 32 ;  *<------ provoquera une erreur de compilation interdisant la modification de la valeur de x.*  

*2.3 Priorité d'opérateurs en Java*

Les 39 opérateurs de Java sont détaillés par famille, plus loin . Ils sont utilisés comme dans tous les langages impératifs pour manipuler, séparer, comparer ou stocker des valeurs. Les opérateurs ont soit une seul opérande, soit deux opérandes, il n'existe en Java qu'un seul opérateur à trois opérandes (comme en C) l'opérateur conditionnel  **" ? : "**.

Les opérateurs de Java sont classés par ordre de priorité croissante (0 est le plus haut niveau, 14 le plus bas niveau). Ceci sert lorsqu'une expression contient plusieurs opérateurs à **indiquer l'ordre dans lequel s'effectueront les opérations**.

* 2**+**3\*4 vaut 14 car l'opérateur \* est plus prioritaire que l'opérateur **+**, donc l'opérateur \* est effectué en premier.
* Lorsqu'une expression contient des opérateurs de **même priorité alors Java effectue les évaluations de gauche à droite**. Par exemple l'expression 12/3\*2 vaut 8 car Java fait la correspondance suivante : ((12/3)\*2).

|  |  |
| --- | --- |
| **priorité** | **opérateurs** |
| 0 | ( )     [  ]      . |
| 1 | !     ~     ++    -- |
| 2 | \*     /    % |
| 3 | +    - |
| 4 | <<    >>    >>> |
| 5 | <     <=    >   >= |
| 6 | = =     != |
| 7 | & |
| 8 | ^ |
| 9 | | |
| 10 | && |
| 11 | || |
| 12 | ? : |
| 13 | =  \*=  /=  %=  +=  -=  ^=  &=  <<=  >>=  >>>=  |= |

**3. Les opérateurs en Java**

*3.1 Opérateurs arithmétiques*

***Ils travaillant avec des opérandes à valeur immédiate ou variable***

|  |  |  |  |
| --- | --- | --- | --- |
| Opérateur | priorité | action | exemples |
| + | 1 | signe positif | +a;  +(a-b); +7 (unaire) |
| - | 1 | signe négatif | -a;  -(a-b); -7 (unaire) |
| \* | 2 | multiplication | 5\*4;  12.7\*(-8.31); 5\*2.6 |
| / | 2 | division | 5 / 2;    5.0 / 2;   5.0 / 2.0 |
| % | 2 | reste | 5 % 2;    5.0 %2;   5.0 % 2.0 |
| + | 3 | addition | a+b; -8.53 + 10; 2+3 |
| - | 3 | soustraction | a-b; -8.53 - 10; 2-3 |

Ces opérateurs sont binaires (à deux opérandes) exceptés les opérateurs de signe positif ou négatif. Ils travaillent tous avec des opérandes de types entiers ou réels. Le résultat de l'opération est converti automatiquement en valeur du type des opérandes.

L'opérateur " **%** " de reste n'est intéressant que pour des calculs sur les entiers longs, courts, signés ou non signés : il renvoie le reste de la division euclidienne de 2 entiers.

***Exemples d'utilisation de l'opérateur de division selon les types des opérandes et du résultat :***

|  |  |  |
| --- | --- | --- |
| ***programme Java*** | ***résultat obtenu*** | ***commentaire*** |
| **int** x = 5 , y  ; | **x = 5 , y =???** | *déclaration* |
| **float** a , b = 5 ; | **b = 5 , a =???** | *déclaration* |
| **y = x / 2 ;** | **y = 2** *// type int* | **int** x et **int** 2  *résultat* **: int** |
| **y = b / 2 ;** | ***erreur de conversion*** | *conversion automatique*  *impossible (****float*** *b -->* ***int*** *y)* |
| **y = b / 2.0 ;** | ***erreur de conversion*** | *conversion automatique*  *impossible (****float*** *b -->* ***int*** *y)* |
| **a = b / 2 ;** | **a = 2.5** *// type float* | **float** b et **int** 2  *résultat* **: float** |
| **a = x / 2 ;** | **a = 2.0** *// type float* | **int** x et **int** 2  *résultat* **: int**  *conversion automatique*  ***int*** *2 -->* ***float****2.0* |
| **a = x / 2f ;** | **a = 2.5** *// type float* | **int** x et **float** 2f  *résultat* **: float** |

Pour l'instruction précédente " **y = b / 2** " engendrant une erreur de conversion voici deux corrections possibles utilisant le transtypage :

y = (**int**)b / 2 ;    *// b est converti en* ***int*** *avant la division qui s'effectue sur deux* ***int****.*   
y = (**int**)(b / 2) ; *// c'est le résultat de la division qui est converti en* ***int****.*

***Rermarque : opérateurs travaillant avec une  unique variable comme opérande***

|  |  |  |  |
| --- | --- | --- | --- |
| **Opérateur** | **priorité** | **action** | **exemples** |
| ++ | 1 | post ou pré incrémentation :  incrémente de 1 son opérande numérique : short, int, long, char, float, double. | ++a;  a++; (unaire) |
| -- | 1 | post ou pré décrémentation :  décrémente de 1 son opérande numérique : short, int, long, char, float, double. | --a;  a--; (unaire) |

L'objectif de ces opérateurs est l'optimisation de la vitesse d'exécution du programme dans la machine virtuelle Java.

**post-incrémentation : k--**

***Exemple 1 :***

|  |  |  |
| --- | --- | --- |
| **int** k = 5 , n ;  **n = k++ - k ;** | **n = -1** | **k = 6** |

Dans l'instruction k++ - k nous avons le calcul suivant : la valeur de k (k=5) est utilisée comme premier opérande de la soustraction, puis elle est incrémentée (k=6), la nouvelle valeur de k est maintenant utilisée comme second opérande de la soustraction ce qui revient à calculer n = 5-6 et donne n = -1 et k = 6.

***Exemple 2 :***

|  |  |  |
| --- | --- | --- |
| ***int k = 5 , n ;    n = k - k++  ;*** | ***n = 0*** | ***k = 6*** |

Dans l'instruction k - k++ nous avons le calcul suivant : la valeur de k (k=5) est utilisée comme premier opérande de la soustraction, le second opérande de la soustraction est k++ c'est la valeur actuelle de k qui est utilisée (k=5) avant incrémentation de k, ce qui revient à calculer n = 5-5 et donne n = 0 et k = 6.

***Exemple 3 :***Utilisation de l'opérateur de post-incrémentation en combinaison avec un autre opérateur unaire. ( = ; ~ ) : cas de l’affectation

**int** nbr1, z , t , u , v  ;

|  |  |  |
| --- | --- | --- |
| nbr1 = 10 ;  **v = nbr1++** | **v = 10** | **nbr1 = 11** |

**pré-incrémentation : ++k**   
la valeur de k est d'abord augmentée de un ensuite utilisée dans l'instruction. ***Exemple1 :***   
**int** k = 5 , n ;

|  |  |  |
| --- | --- | --- |
| **n = ++k ;** | **n = 6** | **k = 6** |

***Exemple 2 :***

|  |  |  |
| --- | --- | --- |
| ***int k = 5 , n ;    n = ++k - k ;*** | ***n = 0*** | ***k = 6*** |

Dans l'instruction ++k - k nous avons le calcul suivant : le premier opérande de la soustraction étant ++k c'est donc la valeur incrémentée de k (k=6) qui est utilisée, cette même valeur sert de second opérande à la soustraction ce qui revient à calculer n = 6-6 et donne n = 0 et k = 6.

***Exemple 3 :***

|  |  |  |
| --- | --- | --- |
| ***int k = 5 , n ;    n = k - ++k ;*** | ***n = -1*** | ***k = 6*** |

Dans l'instruction k - ++k nous avons le calcul suivant : le premier opérande de la soustraction est k (k=5), le second opérande de la soustraction est ++k, k est immédiatement incrémenté  (k=6) et c'est sa nouvelle valeur incrémentée qui est utilisée, ce qui revient à calculer n = 5-6 et donne n = -1 et k = 6.   
    
  **post-décrémentation : k--**   
la valeur de k est d'abord utilisée telle quelle dans l'instruction, puis elle est diminuée de un à la fin.

***Exemple1 :***   
**int** k = 5 , n ;

|  |  |  |
| --- | --- | --- |
| **n = k-- ;** | **n = 5** | **k = 4** |

**pré-décrémentation : --k**   
la valeur de k est d'abord diminuée de un puis ensuite utilisée dans l'instruction.

***Exemple1 :***   
**int** k = 5 , n ;

|  |  |  |
| --- | --- | --- |
| **n = --k ;** | **n = 4** | **k = 4** |

*3.2 Opérateurs de comparaison*

Ces opérateurs employés dans une expression renvoient un résultat de type booléen (false ou true) comme dans n’importe quel langage algorithmique (C, pascal, etc...). Ce sont des opérateurs à deux opérandes.

|  |  |  |  |
| --- | --- | --- | --- |
| **Opérateur** | **priorité** | **action** | **exemples** |
| < | 5 | strictement inférieur | 5 < 2 ;  x+1 < 3 ;   y-2 < x\*4 |
| <= | 5 | inférieur ou égal | -5 <= 2 ;  x+1 <= 3 ;  etc... |
| > | 5 | strictement supérieur | 5 > 2 ;  x+1 > 3 ;  etc... |
| >= | 5 | supérieur ou égal | 5 >= 2 ;    etc... |
| = = | 6 | égal | 5 = = 2 ;  x+1 = = 3 ; etc... |
| != | 6 | différent | 5 != 2 ;  x+1 != 3 ; etc... |

*3.3 Opérateurs booléens*

Ce sont les opérateurs classiques de l'algèbre de boole **{** { **V**, **F** }, ! , & , | **}** où { **V**, **F** } représente l'ensemble {**V**rai,**F**aux}. Les connecteurs logiques ont pour syntaxe en Java :  **! , & , |** :

**&** : { **V**, **F** } x { **V**, **F** } 🡪{ **V**, **F** } *(opérateur* ***binaire*** *qui se lit " et ")*   
**|** : { **V**, **F** } x { **V**, **F** } 🡪{ **V**, **F** } *(opérateur* ***binaire*** *qui se lit " ou ")*   
**!** : { **V**, **F** } 🡪{ **V**, **F** } *(opérateur* ***unaire*** *qui se lit " non ")*

***Table de vérité des opérateurs ( p et q étant des expressions booléennes)***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **q** | **! p** | **p & q** | **p | q** |
| V | V | F | V | V |
| V | F | F | F | V |
| F | V | V | F | V |
| F | F | V | F | F |

Java dispose de 2 clones des opérateurs binaires **&** et **|** . Ce sont les opérateurs **&&** et **||** qui se différentient de leurs originaux **&** et **|** par leur mode d'exécution optimisé

*Exemple :* &&

Si p est faux (p = **F**)  , il est inutile d'évaluer q car l'expression p **&**q est fausse.

*Exemple : | |*

Si p est vrai (p = **V**)  , il est inutile d'évaluer q car l'expression p **|**q est vraie.

***En résumé:***

|  |  |  |  |
| --- | --- | --- | --- |
| **Opérateur** | **priorité** | **action** | **exemples** |
| **!** | 1 | **non** booléen | ! (5 < 2) ;  !(x+1 < 3) ; etc... |
| **&** | 7 | **et** booléen complet | (5 = = 2) & (x+1 < 3) ;   etc... |
| **|** | 9 | **ou** booléen complet | (5 != 2) | (x+1 >= 3) ;   etc... |
| **&&** | 10 | **et** booléen optimisé | (5 = = 2) && (x+1 < 3) ;   etc... |
| **||** | 11 | **ou** booléen optimisé | (5 != 2) || (x+1 >= 3) ;   etc... |

*3.4 Opérateurs bit level*

Ce sont des opérateurs de bas niveau en Java dont les opérandes sont exclusivement l'un des types entiers ou caractère de Java (short, int, long, char, byte). Ils permettent de manipuler directement les bits du mot mémoire associé à la donnée.

12 – La compilation

Java comme pascal, C etc... fait partie de la famille des langages évolués (ou langages de haut niveau) qui ne sont pas compréhensibles immédiatement par le processeur de l'ordinateur. Il est donc nécessaire d'effectuer une "**traduction**" d'un programme écrit en langage évolué afin que le processeur puisse l'exécuter.

Les deux voies utilisées pour exécuter un programme évolué sont la **compilation** ou l'**interprétation** :

* Un compilateur du langage X pour un processeur P, est un logiciel qui traduit un programme source écrit en X en un programme cible écrit en instructions machines exécutables par le processeur P.
* Un interpréteur du langage X pour le processeur P, est un logiciel qui ne produit pas de programme cible mais qui effectue lui-même immédiatement les opérations spécifiées par le programme source.

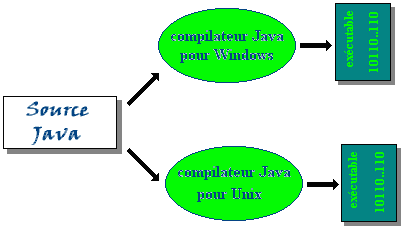
***Un compromis assurant la portabilité d'un langage : une pseudo-machine***

Beaucoup de langages possèdent pour une plate-forme fixée des interpréteurs ou des compilateurs, moins possèdent une p-machine, **Java est l'un de ces langages**. Nous décrivons ci-dessous le mode opératoire en Java.   
 

*12.1 Bytecode et Compilation native*

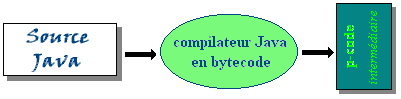
***Compilation native***

La compilation native consiste en la traduction du source java (éventuellement préalablement traduit instantanément en code intermédiare) en langage binaire exécutable sur la plate-forme concernée. Ce genre de compilation est équivalent à n'importe quelle compilation d'un langage dépendant de la plate-forme, l'**avantage est la rapidité d'exécution des instructions machines par le processeur central.**

  
Programe source java : xxx.**java** (*portable*)   
Programe exécutable sous windows : xxx.**exe** (*non portable*)   
    
***Bytecode***

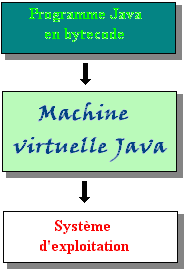
La compilation en bytecode (ou pseudo-code ou p-code ou code intermédiaire) est semblable à l'idée du p-code de N.Wirth pour obtenir un portage multi plate-formes du pascal. Le compilateur **Javac** traduit le programme source xxx.**java** en un code intermédiaire indépendant de toute machine physique et non exécutable directement, le fichier obtenu se dénomme xxx.**class**. Seule une p-machine (dénommée **machine virtuelle java**) est capable d'exécuter ce bytecode. Le bytecode est aussi dénommé **instructions virtuelles java**.

***Figure*** : un programme source ***Exemple.java*** est traduit par le compilateur (dénommé **Javac** ) en un programme cible écrit en bytecode nommé ***Exemple.class***

  
Exemple.java ------------> compilateur Javac ------> Exemple.class

*12.2 La machine virtuelle Java*

Une fois le programme source java traduit en bytecode, la machine virtuelle java se charge de l'exécuter sur la machine physique à travers son système d'exploitation (Windows, Unix, MacOs,...)



Inutile d'acheter une machine virtuelle java, tous les navigateurs internet modernes (en tout cas Internet explorer et Netscape) intègrent dans leur environnement une machine virtuelle java qui est donc installée sur votre machine physique et adaptée à votre système d'exploitation, dès que votre navigateur internet est opérationnel.   
 

*12.4 Premier programme*

**class**  AlgoJavaProg

{   
 static int x = 10, y, z = 2 ;   
 static char carlu = '#' ;   
// etc... *autres déclarations de variables de classe*

**static void main( String[ ] args )**

{ int a, b = -8 ;   
    char c = '@' ;   
  //  etc... *autres déclarations de variables locales*

...... *corps de la méthode principale*

}

static int methode1( ... *paramètres* ... )

{float a = 10.6f ;   
   char b = '?' ;

  } // fin de la focnction  
etc... *autres déclaration de méthodes*

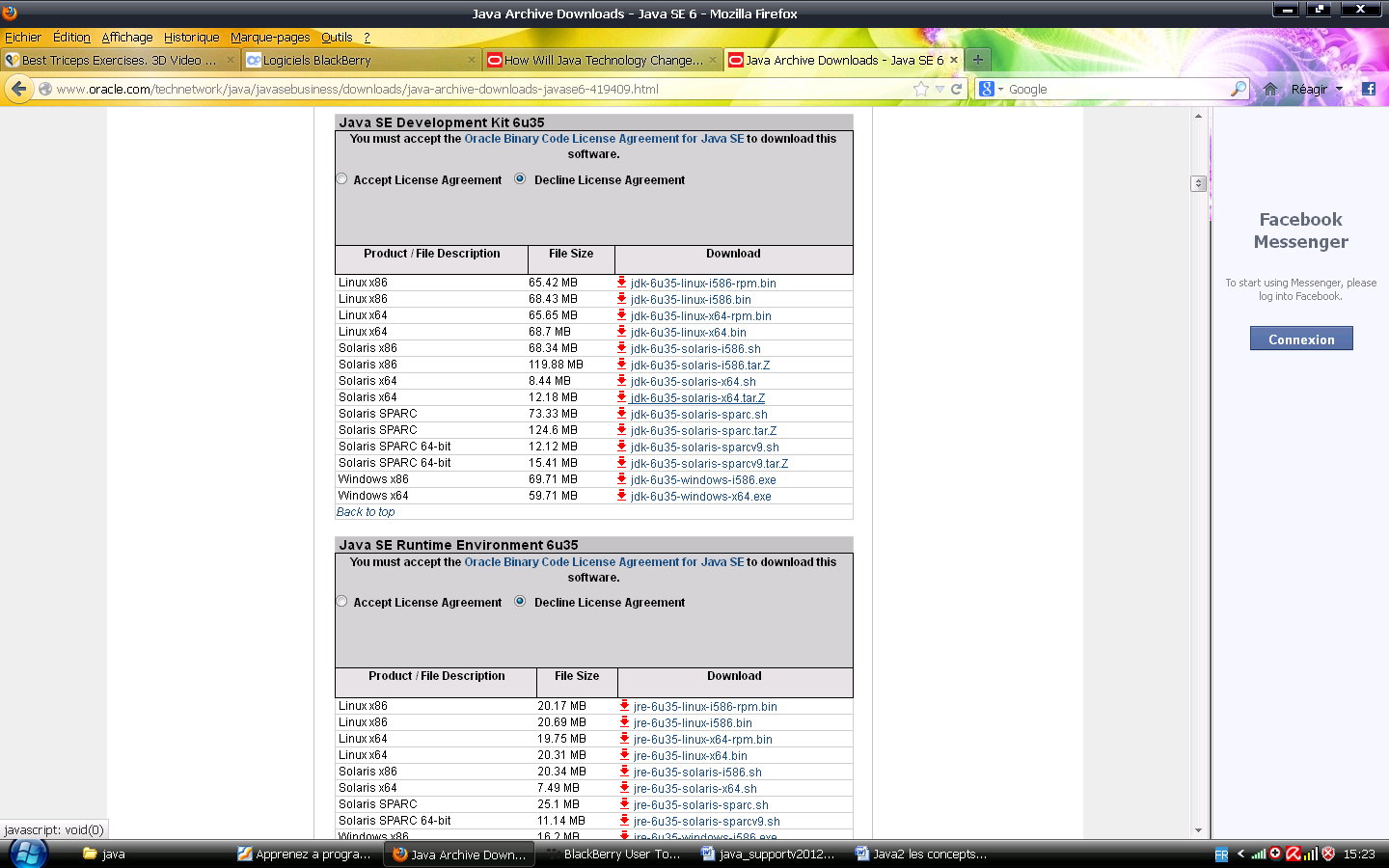
}

Les outils JAVA pour ce cours

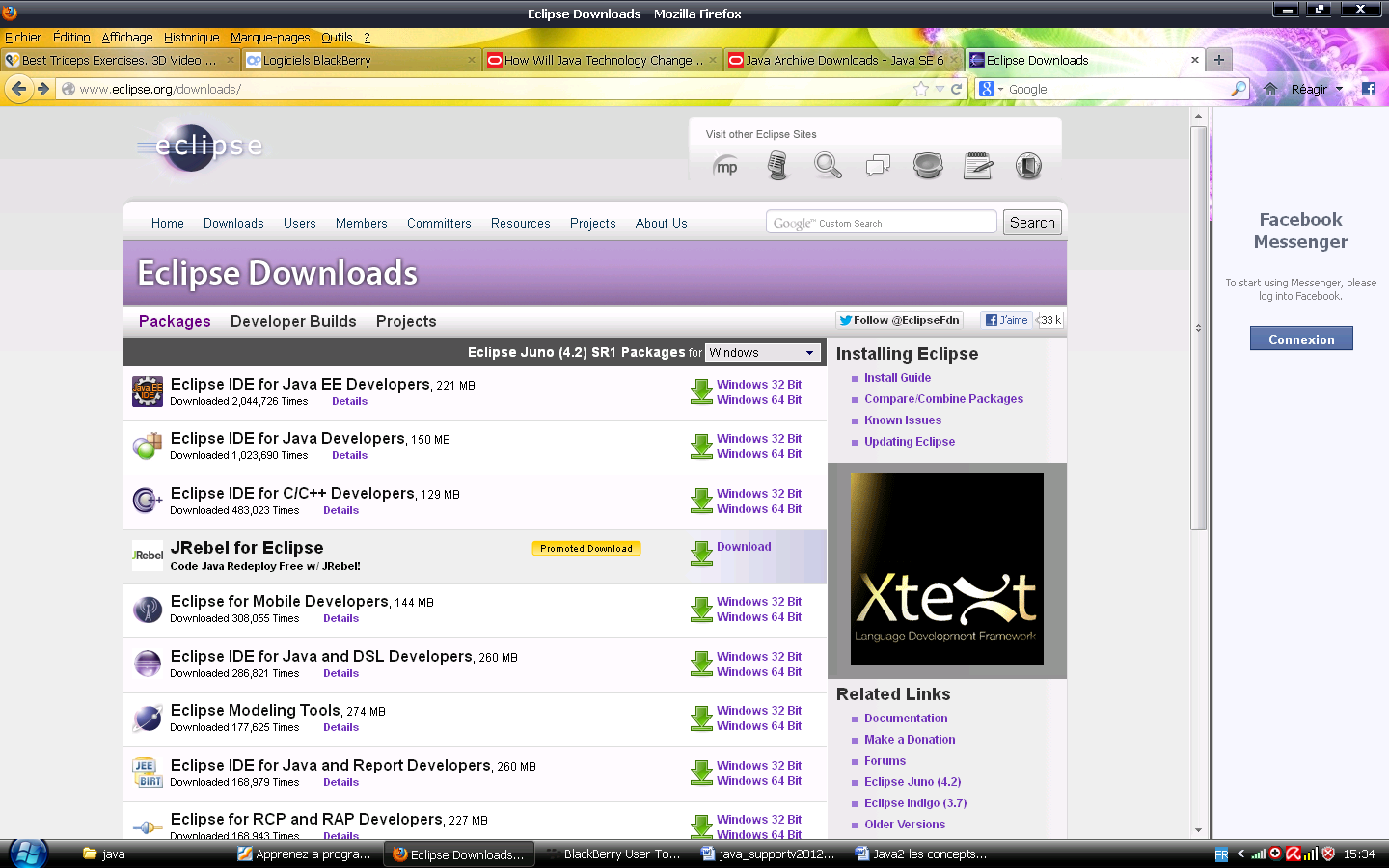
* Java SE 1.6
* Eclipse ou Jdevelopper ou Netbeans

Installation des outils

Télécharger et installer JAVA SE 1.6 sur le site de : [www.oracle.com](http://www.oracle.com). Choisir la version correspondante au système d’exploitation cible.



Télécharger et installer ECLIPSE SDK



Unité :2

Thème : **Java 2**, les instructions et les structures de contrôles

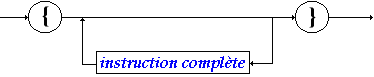
Java s’inspire en grande partie de la syntaxe du langage C ANSI.

**21. instructions-bloc**

* Les commentaires sur une ligne débutent par **//....**
* Les commentaires sur plusieurs lignes sont encadrés par **/\*  ...   \*/**

*2.1.1 instruction :*  Java permet de regrouper entre des accolades plusieurs instructions comme en C ou C++.

*21.2 instruction simple :*instructioncomplete  
Toutes les instructions se terminent en Java par un point-virgule " ; "

*21.3 bloc - instruction composée :* 

*2.1.4 visibilité dans un bloc - instruction  :*

*Exemple de déclarations licites et de visibilité dans 3 blocs instruction imbriqués :*

**int** a, b = 12;   
   **{ int** x , y = 8 ;   
     **{ int** z =12;   
            x = z ;   
            a = x + 1 ;   
        **{ int** u = 1 ;   
            y = u - b ;   
         **}**   
     **}**   
   **}**

**2.2. L'affectation**

Le symbole d'affectation est  " = ", soit l'instruction :   
x = y ;  // *x doit obligatoirement être un identificateur de variable.*

*2.2.1 Affectation simple*

L'affectation peut être utilisée dans une expression :

*soient les instruction suivantes  :*

**int** a , b = 56 ; // a= ??? et b est égal à 56  
a = (b = 12)+8 ;  **//** *b reçoit 12 et a reçoit 20*  
a = b = c = d =8 ; **//** *affectation multiple*

*2.2 Raccourcis et opérateurs d'affectation*

Soit **op** un opérateur  op ∈  { +, - , \* , / , % , << , >> , >>> , & , | , ^ }

Java offre un **raccourci syntaxique** qui permettra une traduction optimisée de « a = a **op** b » .

x **op**= y ;     signifie en fait  :   x = x **op** y ; // déjà vu en C et C++

*soient les instruction suivantes  :*

**int** a , b = 56 ;   
a = -8 ;   
a += b ;  **//** équivalent à : a = a + b   
b \*= 3 ;  **//** équivalent à : b = b \* 3

**2.3.** [**Les conditions**](http://www.univ-tours.fr/discala/LesChapitres.html/Java/Cours1/Chap1.2.htm#2.0#2.0)

*2.3.1 les instructions conditionnelles*

**Syntaxe :**

* **if** ( Expr ) Instr ; // ici instruction 2 est vide
* **if** ( Expr ) Instr1 ; **else** Instr2 ;

L'instruction conditionnelle de java est classiquement celle des langages algorithmiques; comme en pascal  l'expression doit être de type booléen (différent du C).

*Exemple d'utilisation du if..else (*

|  |  |
| --- | --- |
|  | **Java** |
|  | **int** a , b , c ;  ....  **if** ( b = = 0 ) c =1 ;  **else** {  c = a / b;  System.out.println("c = " + c);  }  **if** ((c = a\*b) != 0) c += b;  **else** c = a; |

**Commentaires :**

L'instruction  " **if** ((c = a\*b) != 0) c +=b; **else** c = a; " contient une affectation intégrée.

*2.3.2 L’opérateur conditionnel*

Il s'agit ici comme dans le cas des opérateurs d'affectation d'une sorte de raccourci entre l'opérateur conditionnel **if**...**else** et l'affectation. Le but étant d'optimiser le bytecode engendré.

**Syntaxe :**   
operif

Où expression est une ***expression*** renvoyant une valeur booléenne (le test), ***valeur*** sont des expressions genérales (variable, expression numérique, boolénne etc...) renvoyant une valeur de type T.

*Exemple* :

**int** a,b,c ;   
c =   a = = 0  **?**  b  **:** a+1 ;   
Si l'***expression*** est true l'opérateur renvoie la première valeur, (dans l'exemple c vaut la valeur de b)   
Si l'***expression*** est false l'opérateur renvoie la seconde valeur (dans l'exemple c vaut la valeur de a+1).

Reprise de l'exemple avec un **if**..**else** :

**if** (a = = 0) c = b;   
**else** c = a+1;

question : utiliser l'opérateur conditionnel pour calculer le plus grand de deux entiers.

réponse : **int** a , b , c ; ....   
 c = a>b ? a : b ;

**2.4. Les itérations**

*2.4.1.Itération while*

**Syntaxe :**   
while

Où expression est une ***expression*** renvoyant une valeur booléenne (le test de l'itération).

**Sémantique :**   
Identique à celle du pascal (instruction algorithmique **tantque** .. **faire** .. **ftant**) avec le même défaut de fermeture de la boucle.

***Exemple de boucle while***

**while** **(** Expr **)** Instr ; // **Instr** peut être un bloc d’instructions

*2.4.2.Itération do...while*

**Syntaxe :**   
dowhile

Où expression est une ***expression*** renvoyant une valeur booléenne (le test de l'itération).

**Sémantique :**   
"**do** Instr **while** ( Expr )" fonctionne comme l'instruction algorithmique **répéter**  Instr  **jusquà non** Expr.

***Exemple de boucle do...while***

|  |
| --- |
| **do**  **{** InstrA ;      InstrB ; ...  **} while** **(** Expr **)** |

2.*4.3.Itération* ***for***

**Syntaxe :**   
"**for** (**Expr1** ; **Expr2** ; **Expr3** ) Instr" fonctionne au minimum comme l'instruction algorithmique **pour**... **finpour**.

***Exemples***

|  |  |
| --- | --- |
| **for** ( **i = 1**; **i<=10**; **i++** )  **{** InstrA ;      InstrB ; ...  **}** | **for** ( **i = 10, k = i** ; **i>-450** ; **k += i ,  i -= 15**)  **{** InstrA ;      InstrB ; ...  **}** |

* Le premier exemple montre une boucle for classique.
* Le second exemple montre une boucle toujours contrôlée par une variable "i", mais dont le pas de décrémentation séquentiel est de -15.

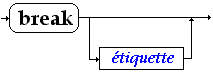
Un exemple de boucle for possédant deux variables de contrôle :

|  |
| --- |
| *//inverse d'une suite de caractère dans un tableau par permutation des extrêmes*  **char** [ ] Tablecar ={'a','b','c','d','e','f'} ;  **for** ( i = 0 , j = 5 ;  i<j ; i++ , j-- ) // les variations de i et j contrôlent la boucle { **char** car ;     car = Tablecar[i]; // la variable **car** joue le rôle du buffer    Tablecar[i ]= Tablecar[j];     Tablecar[j] = car;    } |

**2.5.** [**Break et continue**](http://www.univ-tours.fr/discala/LesChapitres.html/Java/Cours1/Chap1.2.htm#2.0#2.0)

*2.5.1.Interruption break*

**Syntaxe  :**



**Sémantique :**   
Une instruction **break** ne peut se situer qu'à l'intérieur du corps d'instruction d'un bloc **switch** ou de l'une des trois itérations **while**, **do…while**, **for**.

Lorsque **break** est présente dans **while**, **do…while**, **for** :

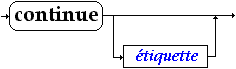
* Si **break** n'est pas suivi d'une étiquette elle interrompt l'exécution de la boucle dans laquelle elle se trouve, l'exécution se poursuit après le corps d'instruction.
* Si **break** est suivi d'une étiquette elle fonctionne comme un **goto** (utilisation **déconseillée** en programmation moderne !)

***Exemple d'utilisation du break dans un for :***   
*(recherche séquentielle dans un tableau)*

|  |
| --- |
| **int** [ ] table= {12,-5,7,8,-6,6,4,78,2};  **int** elt = 4;     **for** ( i = 0 ; i<8 ; i++ )       **if** (elt= =table[i]) **break** ; // *il faut arrêter de chercher* quand on a trouvé    **if** (i = = 8)System.out.println("valeur : "+elt+" pas trouvée.");     **else** System.out.println("valeur : "+elt+" trouvée au rang :"+i); |

*2.5.2.Rebouclage continue*

**Syntaxe  :**



**Sémantique :**   
Une instruction **continue** ne peut se situer qu'à l'intérieur du corps d'instruction de l'une des trois itérations **while**, **do..while**, **for**.

* Si **continue** n'est pas suivi d'une étiquette, elle agit comme si l'on venait d'exécuter la dernière instructions du corps de la boucle.
* Si continue est suivi d'une étiquette elle fonctionne comme un **goto**.

*Exemple d'utilisation du continue dans un for :*

L'instruction **continue** présente dans une telle boucle **for** s'effectue ainsi :

* exécution immédiate de **Expr3 ;**
* ensuite, exécution de **Expr2** ;

réexécution du corps de boucle si la condition de bouclage est respectée.

|  |
| --- |
| //recopie dans le tableau d'entiers **tb** les valeurs non nulles du tableau d'entiers **ta**.  **int** [ ] ta = {12,-5,7,8,-6,6,4,78,2}, tb = **new** **int**[8];  **for** ( **i = 0, n = 0** ; **i<8** ;  **i++ , k = 2\*n** )  { **if** ( ta[i] = = 0 ) **continue** ;    tb[n] = ta[i];    n++;  } |

***Explications***

Si l'expression ( ta[i] = = 0 ) est true, la suite du corps des instructions de la boucle (tb[n] = ta[i];  n++;) n'est pas exécutée et il y a rebouclage du **for** .

Le déroulement est alors le suivant :

* **i++ , k = 2\*n** en premier ;
* puis la condition de rebouclage : **i<8** ;

*Programme équivalent avec* **while**

|  |
| --- |
| **i = 0; n = 0 ;**  **while** ( **i<8** )   { **if** ( ta[i] = = 0 ) **continue** ;     tb[n] = ta[i];     n++;     **i++ ; k = 2\*n;**  } |

**2.6. Switch...case**

**Syntaxe  :**

switch ( expr) {

case v1 : intructions1 break;

case v2 : intructions2 break;

...

default : intructions break;

}

***expr*** est une expression de type **byte**, **char**, **short**, **int** ou **long** et les ***vi*** sont des

valeurs constantes du même type que ***expr***. Cette construction exécute le bloc d’instructions

correspondant à la première valeur ***vi*** égale à ***expr***.

La branche **default** est optionnelle et est exécutée si aucun des ***vi*** n’est égal à ***expr***.

On peut regrouper plusieurs valeurs qui nécessitent le même traitement :

**case u : v : w : intructions break;**

***Exemples :***

|  |  |
| --- | --- |
| **Java - source** | **Résultats de l'exécution** |
| **int**  x = 10;  **switch** (x+1)     { **case** 11 : System.out.println(">> case 11");        **case** 12 : System.out.println(">> case 12");         **default** : System.out.println(">> default");      } | >> case 11  >> case 12  >> default |
| **int**  x = 11;  **switch** (x+1)     { **case** 11 : System.out.println(">> case 11");        **case** 12 : System.out.println(">> case 12");         **default** : System.out.println(">> default");      } | >> case 12  >> default |

Après que (x+1) soit évalué, selon sa valeur (11 ou 12) le programme va se dérouter vers **case** 11 ou **case** 12 et continue en séquence (suite des instructions du **bloc switch**)

Utilisée telle quelle, cette instruction n'est pas structurée et donc son utilisation n'est pas conseillée sous cette forme. Par contre elle est très souvent utilisée avec l'instruction **break** afin de simuler le comportement de l'instruction structurée **case**..**of** du pascal

***Exemple de switch..case..break***

|  |  |
| --- | --- |
| **Java - source** | **Résultats de l'exécution** |
| **int**  x = 10;  **switch** (x+1)     { **case** 11 : System.out.println(">> case 11");                       break;        **case** 12 : System.out.println(">> case 12");                        break;        **default** : System.out.println(">> default");      } | >> case 11 |

**2.7.** [**try...catch**](http://www.univ-tours.fr/discala/LesChapitres.html/Java/Cours1/Chap1.2.htm#2.0#2.0)

Java dispose d'un mécanisme puissant présent dans les langages robustes Ada, C++, Delphi pour intercepter et traiter les erreurs ou incidents survenant dans un bloc d'instruction. L'instruction **try...catch** sert à traiter de tels incidents, elle dispose d'un bloc **try** (pour les instructions à protéger)et d'un ou plusieurs blocs **catch** (pour divers genres de traitement en cas d'incidents).

**try** {   
   BLOC à protéger   
  }   
   **catch** (TypeErreur1  Err1) {   
      TRAITEMENT d'une erreur du type TypeErreur1   
  }   
**catch** (TypeErreur2  Err2) {   
      TRAITEMENT d'une erreur du type TypeErreur2   
  }   
...   
**catch** (TypeErreurX  ErrX) {   
      TRAITEMENT d'une erreur du type TypeErreurX   
  }

***Exemple de try...catch*** pour une erreur dans la division de l'entier x par l'entier y :

|  |
| --- |
| **int** x , y ;  **try** {      x = 1;      y = 0;     System.out.println( "division ="+x/y);    }   **catch** (ArithmeticException Err) {     System.out.println("Calcul impossible.");     System.out.println(Err.getMessage());    } |

**2.8 Entrées/sorties**

Java offre deux sortes de communications avec l’extérieur :

• des entrées-sorties traditionnelles : lectures et écritures de textes sur clavier/

écran ou sur fichiers,

**• des interactions à travers un système de fenêtrage : création de fenêtres, création**

de boîtes de dialogues, réaction à des événements.

Pour les entrées-sorties **écran/clavier**, on dispose des appels de méthodes suivants :

**System.out.print(String s);**  impression de la chaîne s,

**System.out.println(String s);** idem avec retour à la ligne.

La lecture des entrées clavier se fait via l'objet Scanner. Ce dernier se trouve dans le package **java.util** qu’il faut importer.

Pour pouvoir récupérer ce vous allez ta er dans la console, vous devrez initialiser

l'objet Scanner avec l'entrée standard, System.in.

Il y a une méthode de récupération de données pour chaque type (sauf les char) :

**nextLine()** pour les String, **nextInt()** pour le s int.

|  |
| --- |
| **import** java.util.Scanner;  public class Main {  public static void main(String[] args){  Scanner sc = new Scanner(System.in);  System.out.println("Saisissez un entier : ");  int i = sc.nextInt();  System.out.println("Saisissez une chaîne : ");  //On vide la ligne avant d'en lire une autre  sc.nextLine();  String str = sc.nextLine();  System.out.println("FIN ! ");  }  } |

Unité :3

Thème : **Java 2**, Les méthodes (fonctions)

**3.1. Les méthodes sont des fonctions**

Les méthodes (ou fonctions) représentent une encapsulation des instructions qui déterminent le fonctionnement d'une classe. Sans méthodes pour agir une classe ne fait rien de particulier, dans ce cas elle ne fait que contenir des attributs.

*31.1 Méthode élémentaire de classe*

Java distingue deux sortes de méthodes : les **méthodes de classe** et les **méthodes d'instance. (***cf. Unité 5***:** la programmation orientée objet)  
Une méthode de classe commence **obligatoirement** par le mot clef   **static**

31*.2 Déclaration d'une méthode*

Comme dans les langages C et Delphi, une fonction comporte une **en-tête** avec des paramètres formels **et un corps de fonction** (ou de méthode ) qui contient les instructions de seront exécutés lors de l’appel.

**Syntaxe :**   
<qualificateurs><type résultat><nom fonction>(<liste paramètres formels>) *// en- tête*

{corps de fonction…}

*exemple : fonction retournant le plus grand de x et y*

boolean superieur (float x, float y)

{return x>y;}

Les **qualificateurs** sont des mots clef permettant de modifier le fonctionnement ou la visibilité d'une méthode. Ce mot clef peut être omis.

Une méthode peut renvoyer un **résultat** d'un type Java quelconque en particulier d'un des types élémentaires (**int, byte, short, long, boolean, double, float, char**) et aussi un résultat de type **objet**. Ce mot ne doit pas être omis. On utilisera le mot clé **void** pour les fonctions qui ne retourne pas de résultat. (les procédures)

La liste des **paramètres** formels est semblable à la partie déclaration de variables en Java (sans initialisation automatique). La liste peut être vide.

Le **corps** de fonction est identique au bloc instruction Java déjà défini auparavant. Le corps de fonction peut être vide (la méthode ne représente alors aucun intérêt).

Exemples d'en-tête de méthodes ***sans paramètres*** en Java

**int** calculer( ){.....} // renvoie un entier de type **int**

**void** uncalcul( ){.....} // procédure **ne renvoyant rien**

Exemples d'en-tête de méthodes ***avec paramètres*** en Java

**int** calculer(**byte** a, **byte** b, **int** x ) {.....} // fonction à 3 paramètres

**boolean** tester( **int** k) {.....} //fonction à 1 paramètre

**void** uncalcul(**int** x, **int** y, **int** z ) {.....} // procédure à 3 paramètres

*31.3 Appel d'une méthode*

L'**appel de méthode** en Java s'effectue très classiquement avec des paramètres effectifs dont le **nombre** doit obligatoirement être le **même** que celui des paramètres formels et le **type** doit être soit le **même**, soit un type **compatible** ne nécessitant pas de transtypage.

Exemple d'appel de méthode-procédure ***sans paramètres*** en Java

**class AppelFonction**   
     { **static** **void** main(String[ ] args)   
        {  afficher( ); //*Appel de la méthode afficher*  
        }

       **static** **void** afficher( ) // *définition de la fonction*  
      { System.out.println("Bonjour");   
      }

    }

Exemple d'appel de méthode-procédure ***avec paramètres de même type*** en Java

|  |
| --- |
| **class** TabRecherche  **{**  **static** **void** main(String[ ] args)       **{** *// recherche séquentielle dans un tableau*          **int** [ ] table= {12,-5,7,8,-6,6,4,78,2};          **long** elt = 4;          **int** i ;          **for** ( i = 0 ; i<=8 ; i++ )          **if** (elt= =table[i]) **break** ;          afficher(**i**,**elt**);        **}**     **static** **void** afficher (**int** rang , **long** val)     **{** **if** (rang == 8)          System.out.println("valeur : "+val+" pas trouvée.");        **else**          System.out.println("valeur : "+val+" trouvée au rang :"+ rang);     **}**  **}** |

afficher(**i**,**elt**); //Les deux paramètres effectifs "**i**" et "**elt**"  sont du même type que le paramètre formel associé.

**3.2** [**Transmission des paramètres en Java**](http://www.univ-tours.fr/discala/LesChapitres.html/Java/Cours1/Chap1.3.htm#2.0#2.0)

*32.1 Compatibilité des types des paramètres*

|  |  |
| --- | --- |
| ***Les types entiers compatibles :***  entiercompatible | ***Les types réels compatibles :***  reelcompatible |

Exemple d'appel de la même  méthode-procédure ***avec paramètres de type compatibles*** en Java

|  |
| --- |
| **class** Application5  **{**  **static** **void** main(String[ ] args)       **{** *// recherche séquentielle dans un tableau*          **int** [ ] table= {12,-5,7,8,-6,6,4,78,2};          **byte** elt = 4;          **short** i ;          **for** ( i = 0 ; i<8 ; i++ )          **if** (elt= =table[i]) **break** ;          afficher(**i**,**elt**);        **}**     **static** **void** afficher (**int** rang , **long** val)     **{** **if** (rang == 8)          System.out.println("valeur : "+val+" pas trouvée.");        **else**          System.out.println("valeur : "+val+" trouvée au rang :"+ rang);     **}**  **}** |

**Appel de la méthode afficher**

afficher(**i**,**elt**);

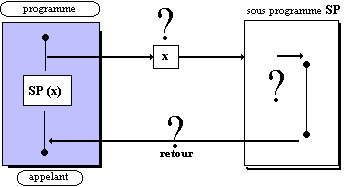
Les deux paramètres effectifs "**i**" et "**elt**"  sont d'un type compatible avec celui du paramètre formel associé.

- Le paramètre effectif  "**i**" est associé au paramètre formel **rang**.(**short** = entier signé sur 16 bits et **int** = entier signé sur 32 bits)

- Le paramètre effectif "**elt**" est associé au paramètre formel **val**.(**byte** = entier signé sur 8 bits et **long** = entier signé sur 64 bits)

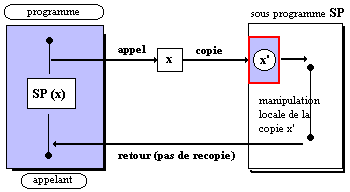
*3.22 Les deux modes de transmission des paramètres*

Un paramètre effectif transmis au sous-programme appelé est en fait un moyen d’utiliser ou d’accéder à une information appartenant au bloc appelant.

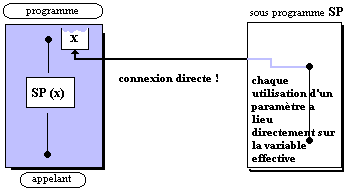


En Java, il existe deux modes de transmission (ou de passage) des paramètres :

* Le passage par **valeur** uniquement réservé à tous les types élémentaires (**int**, **byte**, **short**, **long**, **boolean**, **double**, **float**, **char**).



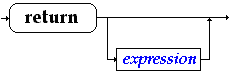
* Le passage par **référence** uniquement réservé à tous les types objets.( cf POO)



Le choix de **passage selon les types** élimine les inconvénients dus à l'encombrement mémoire et à la lenteur de recopie de la valeur du paramètre par exemple dans un passage par valeur, nous verrons plus loin que les **tableaux en Java sont des objets** et qu'ils sont donc passés **par référence**.

*32.3 Les retours de résultat de méthode-fonction*

Les méthodes en Java peuvent renvoyer un résultat de n'importe quel type élémentaire ou objet. Bien qu'une méthode ne puisse renvoyer qu'un seul résultat, l'utilisation du passage par référence d'objets permet aussi d'utiliser les paramètres de type objet d'une méthode comme des variables d'entrée/sortie. En Java le retour de résultat est passé grâce au mot clé **return** placé n'importe où dans le corps de la méthode.

**Syntaxe :**   


*Return « expression »* est obligatoire et de même type que la donnée retournée par la méthode*.*

Seulement les méthodes qui ne retourne pas de donnée ( procédures ) peuvent se passer de l’expression return

***Exemple la fonction f(x)=3x-7***

|  |
| --- |
| **class** Fonction  **{**  **static** **void** main(String[ ] args)       **{** *// ...*          **int** x , y ;          x = 4 ;          y = f(5) ; //*Appel de la méthode f f ( 5 ) ;*         y = f(x) ; // *un autre appel de la fonction*         System.out.println("f(x)="+ f(x) );          System.out.println("f(5)="+ f(5) );        **}**     **static** **int f** (**int** x )     **{**  **return** 3\*x-7;     **}**  **}** |

Dans les deux cas la valeur 5 ou la valeur 4 de x est recopiée dans la **zone de pile (**cf. unité 1**)** de la machine virtuelle java.

**33. Visibilité des variables**

Les variables en Java sont visibles (donc utilisables) dans le bloc dans lequel elles ont été définies.

*33.1 Visibilité de bloc*

Java est un langage à structure de blocs ( comme pascal et C ) dont le principe général de visibilité est :

Toute variable déclarée dans un bloc est visible dans ce bloc et dans tous les blocs imbriqués dans ce bloc. En java les blocs sont constitués par :

* les classes,
* les méthodes,
* les instructions composées,
* les corps de boucles,
* les try...catch.

Le masquage des variables n'existe que pour les variables déclarées dans des méthodes :

Il est **interdit de redéfinir** une variable déjà déclarée dans une méthode soit :

* comme paramètre de la méthode,
* comme variable locale à la méthode,
* dans un bloc inclus dans la méthode.

Il est **possible de redéfinir**  une variable déjà déclarée dans une classe.

*32.2 Variables dans une classe, dans une méthode*

Les variables définies (déclarées, et/ou initialisées) dans une classe sont accessibles à toutes les méthodes de la classe, la visibilité peut être modifiée par les qualificateurs **public** ou **private** (cf. unité 5 POO)

***Exemple de visibilité dans une classe***

|  |
| --- |
| **class** ExempleVisible1 **{**    **int** **a** = 10;  **int** g (**int** x )    **{**  **return**  3\*x-**a**;  // **a** *est visible*   **}**  **int** f (**int** x, **int** **a** )    **{**  **return** 3\*x-**a**;  // **a** *est visible mais masque par le redéfinition*    **}**  **}** |

Les variables définies dans une méthode subissent les règles classiques de la visibilité du bloc dans lequel elles sont définies :

Elles sont visibles dans toute la méthode et dans tous les blocs imbriqués dans cette méthode et seulement à ce niveau (les autres méthodes de la classe ne les voient pas), c'est pourquoi on emploie aussi le terme de variables locales à la méthode.

***Exemple de variables locales***

|  |  |
| --- | --- |
| **class** ExempleVisible3 **{**    **static int a** = 10;  **static int** g (**int** x )    **{**  **char** car = 't';         **long** **a** = 123456;         ....  **return**  3\*x-**a**;     **}**  **static int** f (**int** x, **int** **a** )    **{**  **char** car ='u';         ....  **return** 3\*x-**a**;     **}**  **}** | La variable de classe "a" définie dans **static int** **a** = 10; est masquée dans les deux méthodes **f** et **g**.  Dans la méthode **g**, c'est la variable locale **long a** = 123456 qui masque la variable de classe **static int** **a**. **char** car ='t'; est une variable locale à la méthode **g**.  - Dans la méthode **f**, **char** car ='u'; est une variable locale à la méthode **f**, le paramètre **int a** masque la variable de classe **static int** **a**.  Les variables locales **char** car n'existent que dans la méthode où elles sont définies, les variables "car" de **f** et celle de **g** n'ont aucun rapport entre elles, bien que portant le même nom. |

*4.3 Variables dans un bloc autre qu'une classe ou une méthode*

Les variables définies dans des blocs du genre instructions composées, boucles, try..catch ne sont visibles que dans le bloc et ses sous-blocs imbriqués, dans lequel elle sont définies.

***Exemple correct de variables locales***

|  |  |
| --- | --- |
| **class** ExempleVisible4 **{**  **static int a** = 10, **b** = 2;  **static int** f (**int** x )  **{**  **char** car = 't';  **for** (**int** **i** = 0;  **i** < 5 ;  **i**++)        **{int a**=7;  **if** (**a** < 7)            **{int b** = 8;             **b** = 5-**a**+**i\*b**;            **}**  **else b** = 5-**a**+**i**\***b**;         **}**  **return**  3\*x-**a**+**b**;     **}**  **}** | La variable de classe "a" définie dans **static int** **a** = 10; est masquée dans la méthode **f**  dans le bloc imbriqué **for**.  La variable de classe "b" définie dans **static int** **b** = 2; est masquée dans la méthode **f**  dans le bloc imbriqué **if**.  Dans l'instruction **{** **intb** = 8; **b** = 5-**a**+**i\*b**; **}** , c'est la variable **b** interne à ce bloc qui est utilisée car elle masque la variable **b** de la classe.  Dans l'instruction **else** **b** = 5-**a**+**i**\***b**; , c'est la variable **b** de la classe qui est utilisée (car la variable **int b** = 8 n'est plus visible ici) . |

***Exemple de variables locales générant une erreur***

|  |
| --- |
| **class** ExempleVisible5 **{**  **static int a** = 10, **b** = 2;  **static int** f (**int** x )  **{**  **char** car = 't';  **for** (**int** **i** = 0;  **i** < 5 ;  **i**++)         **{int a**=7;  **if** (**a** < 7)             **{int b** = 8, **a = 9**;              **b** = 5-**a**+**i\*b**;             **}**  **else b** = 5-**a**+**i**\***b**;          **}**  **return**  3\*x-**a**+**b**;     **}**  **}** |

Java produit une erreur de compilation **int b** = 8, **a = 9**; sur la variable **a**, en indiquant que c'est une ***redéfinition*** de variable à l'intérieur de la méthode **f**, car nous avions déjà défini une variable **a**  (**{** **int** **a**=7;...) dans le bloc englobant **for {...}**.

**Remarque :**

Tout en respectant à l'intérieur d'une méthode le principe de visibilité de bloc, Java n'accepte pas le masquage de variable à l'intérieur des blocs imbriqués.

Unité :5

Thème : Java 2 et la programmation orientée objet ( POO )

**5.1 Introduction à la programmation orientée objet**

*5.1.1 Définitions*

La programmation par objets consiste essentiellement à structurer les applications selon les types de données.

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

Ceci conduit à une **modularité** basée sur les types de données. Les traitements sont toujours définis à propos de la définition d’un type de données, et non pas isolément comme c’est le cas pour une modularité basée en priorité sur la décomposition des traitements.

Exemple :

(1) Pour une gestion de réseau ferroviaire, le programmeur par objets sera tenté de définir les objets suivants : trains, voitures, lignes, gares, ... Pour chaque type d’objet, il définira les propriétés de ces objets et les actions que l’on peut leur faire subir : un train est formé d’une collection de voitures, on peut lui accrocher une voiture, lui décrocher une voiture, chaque voiture a une capacité ...

(2) Considérons un robot (simpliste) qui occupe une certaine position (X,Y), il a une orientation parmi {Nord, Est, Sud, Ouest}, il est initialisé avec une position et une orientation données, il peut tourner à droite, il peut avancer d’un pas ... On le décrira par une association de variables d’état et de procédures.

|  |  |
| --- | --- |
| **orientation** | *Les procédures*  Avancer()  tournerADroite()  Initialisations() |

C’est une telle association de données et de procédures que l’on nomme habituellement

un **objet**.

*5.1.2 Les langages orientés objet ( LOO)*

Il existe de nombreux langages permettant la programmation par objets : **Simula**,

**Smalltalk**, **Eiffel**, **Turbo Pascal Objet**, **C++, Java**. Ces langages ont en commun les

notions suivantes :

**•** La notion d’***objet***, qui possède deux aspects totalement indépendants :

**-** Un objet est une ***chose dotée d’une identité*** (on peut le référencer). Ceci s’oppose à la notion de ***valeur*** (par exemple 12 ***n’est pas un objet***).

**-** Un objet peut, accessoirement, être composé de données et de procédures liées à ces données : les données de l’objet constituent le contexte de travail pour les procédures associées.

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

**•** La notion d’***héritage( la hierachie des classes)*** : l’héritage permet de définir des types d’objets plus précis à partir de types d’objets plus généraux. . Elle permet de définir une hiérarchie de types, des plus généraux aux plus spécialisés. L’héritage et les notions qui lui

sont associées, notamment les procédures virtuelles, permettent la conception de logiciels

extensibles et favorisent la réutilisation de modules logiciels existants.

Nous étudierons les notions de programmation par objets du langage **Java**.

**5.2. Les classes**

*5.2.1 Déclaration d'une classe*

En Java , la classe avec ses attributs et ses méthodes sont déclarés et implémentés à un seul endroit.

**class** Robot {

**public** **static** **final** int Nord = 1; // declaration de constantes

public static final int Est = 2;

public static final int Sud = 3;

public static final int Ouest = 4;

public int X; (variables d’état)

public int Y;

public int orientation;

public Robot(int x, int y, int o) { (initialisation)

X=x; Y=y; orientation=o;

}

public Robot() { (autre initialisation possible)

X=0; Y=0; orientation=Nord;

}

public void avancer() { switch (orientation) {

case Nord : Y=Y+1; break;

case Est : X=X+1; break;

case Sud : Y=Y-1; break;

case Ouest: X=X-1; break;

};

}

public void tournerADroite(){

switch (orientation) {

case Nord : orientation=Est ; break;

case Est : orientation=Sud ; break;

case Sud : orientation=Ouest; break;

case Ouest: orientation=Nord ; break;

};

}

}

L’attribut **public**, appliqué à un composant quelconque, donnée ou méthode, signifie que ce composant est désignable par son identificateur depuis le texte de toute autre classe.

*Un exemple de programme contenant deux classes*

//Fichier application.java

public class application  
{  
class compteur {  
int i;  
compteur(){i=0;}  
void incr(){i++;}  
boolean egal(int j)  
 {return(i==j);}  
}  
  
class FlipFlop {  
 String etat;  
 FlipFlop() {etat="flip";}  
 void change() {  
 if ( etat.equals("flip")) {etat="flop";}  
 else {etat="flip";}  
 }  
}  
}

// Fichier TestFlipFlop.java  
import application; class TestFlipFlop {  
static FlipFlop ff=new FlipFlop();  
public static void main(String[] args)  
 { compteur cpt=new compteur();  
 while (!cpt.egal(4)) {  
 System.out.println(ff.etat);  
 ff.change();  
 cpt.incr();  
 }  
 }  
}

*5.2.2 Toutes les classes ont le même ancêtre :* ***héritage***

Toutes les classes Java dérivent automatiquement d'une seule et même classe ancêtre : la classe **Object**. En java le mot-clef pour indiquer la dérivation (héritage) à partir d'une autre classe est le mot **extends**, lorsqu'il est omis c'est donc que la classe hérite automatiquement de la classe **Object** :

|  |
| --- |
| **class** Exemple **extends** Object  **{**   .......  **}** |
| **class** Exemple   **{**    .......  **}** |

L'héritage en Java est t l'**héritage simple** . Une classe fille qui dérive d'une seule classe mère, hérite de sa classe mère toutes ses méthodes et tous ses champs. En Java la syntaxe de l'héritage fait intervenir le mot clef **extends**, comme dans "**class** Exemple **extends** Object".

Une déclaration du type :   
**class** ClasseFille **extends** ClasseMere {   
}   
signifie que la classe ClasseFille dispose de tous les attributs et les méthodes de la classe ClasseMere.

***Héritage en Java :***

|  |
| --- |
| **class** ClasseMere  {     // champs de ClasseMere    // méthodes de ClasseMere  }  **class** ClasseFille **extends** ClasseMere  **{**    // hérite des champs de ClasseMere    // hérite des méthodes de ClasseMere  **}** |

Une classe fille peut définir de nouveaux champs et de nouvelles méthodees qui lui sont propres.   
    
5.2.3 *Encapsulation des classes*

En Java depuis le JDK 1.1, on a la possibilité d'***imbriquer*** des classes dans d'autres classes, par conséquent la ***visibilité de bloc s'applique aussi aux classes.*** (cf 5.2.1)

**Mots clefs pour la protection des classes et leur visibilité** . Il y a 2 mots clef possibles pour modifier le comportement d'une classe : **public** et **abstract**.

|  |  |
| --- | --- |
| **Java** | **Signification** |
| mot clef **abstract** :  **abstract** **class** ApplicationClasse1 { ... } | classe abstraite **non instanciable**. Aucun objet ne peut être créé. |
| mot clef **public** :  **public** **class** ApplicationClasse2 { ... } | classe visible **par n'importe quel** programme, elle doit avoir le même nom que le fichier de bytecode xxx.class qui la contient |
| pas de mot clef :  **class** ApplicationClasse3 { ... } | classe visible **seulement** par toutes les autres classes du module où elle est définie. |

5.2.4 *Exemple de classe inclue dans un package*

***Le package :***

|  |  |
| --- | --- |
| **Java** | **Explication** |
| **package** Biblio;  **abstract class** ApplicationClasse1 { ... }  **public class** ApplicationClasse2 { ... }   **class** ApplicationClasse3 { ... } | Ces 3 "sous-classes" font partie du package Biblio, elles sont visibles par importation séparée (comme précédemment) ou globale du package. |

***Remarques pratiques :***

Pour pouvoir utiliser dans un programme, une classe définie dans un module (**package**) celle-ci doit obligatoirement avoir été déclarée dans le **package**, avec le modificateur **public**.

Pour accéder à la classe Cl1 d'un **package** Pack1, il est nécessaire d'importer cette classe :   
**import** Pack1.Cl1;

L’utilisation de toutes les classes du package se fait par l’instruction**import** Biblio.\* ;

*Les packages du JDK*

Le JDK fournit un grand nombre de classes regroupées en packages nommés java.\*.

**Le package java.lang**

Le package java.lang est chargé automatiquement, ses classes sont donc toujours utilisables.

On y trouve, entre autres :

* la classe Object dont dérivent toutes les autres classes
* les classes représentant les types numériques de bases : Boolean, Byte, Double, Float, Integer, Long
* la classe Math qui fournit des méthodes de calcul des fonctions usuelles en mathématiques
* les classes Character, String et StringBuffer pour la gestion des caractères et des chaînes de caractères
* la classe System que nous utilisons pour afficher du texte sur la console DOS.

**Les autres packages**

Les autres packages doivent être déclarés (mot clé import) pour pouvoir être utilisés.

Nous aurons à étudier entre autres, les packages

* java.awt pour la création d'interfaces graphiques
* java.awt.event pour la gestion des évènements
* java.util pour certaines structures de données
* java.io pour la gestion des fichiers
* java.net pour le téléchargement de fichiers

La documentation du JDK est indispensable pour connaître le contenu de ces classes.

1 class point  
 2 {int x=0,y=0;  
 3   
 4 void move(int dx,int dy)  
 5 {x=x+dx;  
 6 y=y+dy;  
 7 }  
 8   
 9 void affiche()  
10 {System.out.println(x+" "+y);  
11 }  
12   
13 public static void main(String[] args)  
14 {point a;  
15 a = new point();  
16 a.affiche();  
17 a.move(5,6);  
18 a.affiche();  
19 }  
20 }*1.7 Méthodes abstraites*

Le mot clef **abstract** est utilisé pour représenter **une classe ou une méthode abstraite**. Quel est l'intérêt de cette notion ? Avoir des modèles génériques permettant de définir ultérieurement des actions spécifiques.

**Une méthode déclarée en abstract dans une classe mère :**

* + N'a pas de corps de méthode.
  + N'est pas exécutable.
  + Doit obligatoirement être redéfinie dans une classe fille.

Une méthode **abstraite** n'est qu'une **signature** de méthode sans implémentation dans la classe.

***Exemple de méthode abstraite :***

**class** Etre\_Vivant **{**   
**}**

La classe Etre\_Vivant est une classe mère générale pour les êtres vivants sur la planète.

|  |
| --- |
| **class** Serpent **extends** Etre\_Vivant **{**  **}**  **class** Oiseau **extends** Etre\_Vivant **{**  **}**  **class** Homme **extends** Etre\_Vivant **{**  **}** |

Tous ces êtres se déplacent d'une manière générale donc une méthode SeDeplacer est commune à toutes les classes dérivées, toutefois chaque espèce exécute cette action d'une manière différente et donc on ne peut pas dire que se déplacer est une notion concrète mais une notion abstraite que chaque sous-classe précisera concrètement.

|  |
| --- |
| **abstract class** Etre\_Vivant **{**  **abstract** **void** SeDeplacer( );  **}**  **class** Serpent **extends** Etre\_Vivant **{**  **void** SeDeplacer( ) **{**      //.....en rampant    **}**  **}**  **class** Oiseau **extends** Etre\_Vivant **{**  **void** SeDeplacer( ) **{**      //.....en volant    **}**  **}**  **class** Homme **extends** Etre\_Vivant **{**  **void** SeDeplacer( ) **{**      //.....en marchant    **}**  **}** |

*1.8 Classe abstraite,  Interface*

*Classe abstraite*   
Comme nous venons de le signaler une classe Java peut être précédée du mot clef **abstract**, ce qui signifie alors que cette classe est abstraite, nous avons les contraintes de définition suivantes pour une classe abstraite en Java :

Si une classe contient au moins une méthode **abstract**, elle doit impérativement être déclarée en classe **abstract** elle-même. C'est ce que nous avons écrit au paragraphe précédent pour la classe Etre\_Vivant que nous avons déclarée **abstract** parce qu'elle contenait la méthode abstract SeDeplacer.

Une classe **abstract** ne peut pas être instanciée directement, seule une classe dérivée (sous-classe) qui redéfinit obligatoirement toutes les méthodes **abstract** de la classe mère peut être instanciée.

Conséquence du paragraphe précédent, une classe dérivée qui redéfinit toutes les méthodes **abstract** de la classe mère sauf une (ou plus d'une) ne peut pas être instanciée et subit la même règle que la classe mère : elle contient au moins une méthode abstraite donc elle aussi une classe abstraite et doit donc être déclarée en **abstract**.

Une classe **abstract**  peut contenir des méthodes non abstraites et donc implantées dans la classe. Une classe **abstract**  peut même ne pas contenir du tout de méthodes abstraites, dans ce cas une classe fille n'a pas la nécessité de redéfinir les méthodes de la classe mère pour être instanciée.

*Interface*   
Lorsqu'une classe est déclarée en  **abstract** et que toutes ses méthodes sont déclarées en **abstract**, on appelle en Java une telle classe une **Interface**.

**5.3 Les objets : des références**

Les classes sont des descripteurs d'objets, les objets sont les agents effectifs et "vivants" implantant les actions d'un programme. Les objets dans un programme ont une vie propre :

* + Ils naissent (ils sont créés ou alloués).
  + Ils agissent (ils s'envoient des messages grâce à leurs méthodes).
  + Ils meurent (ils sont désalloués, automatiquement en Java).

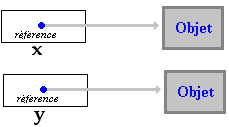
C'est dans le segment de mémoire de la machine virtuelle Java que s'effectue l'allocation et la désallocation d'objets. Le principe d'allocation et de représentation des objets en Java est identique à celui de Delphi il s'agit de la référence, qui est une encapsulation de la notion de pointeur.   
    
  *2.1 Modèle de la référence et machine Java*

Rappelons que dans le modèle de la référence chaque objet (représenté par un identificateur de variable) est caractérisé par un couple (référence, bloc de données). Comme en Delphi, Java décompose l'**instanciation** (allocation) d'un objet en deux étapes :

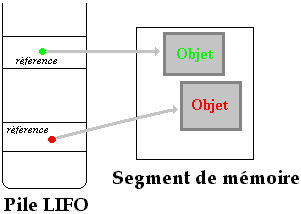
* + La déclaration d'identificateur de variable typée qui contiendra la référence,
  + la création de la structure de données elle-même (bloc objet de données) avec **new**.

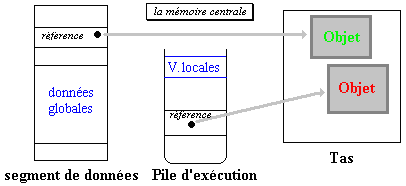
|  |
| --- |
| **Java** |
| **class** **Un**   **{** ...   **}**  *// la déclaration :*  **Un x , y ;**  ....  *// la création :*  **x** = **new Un**( );  **y** = **new Un**( ); |

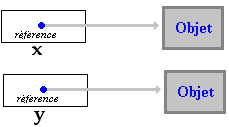
Après exécution du pseudo-programme précédent, les variables x et y contiennent chacune une référence (adresse mémoire) vers un bloc objet différent:

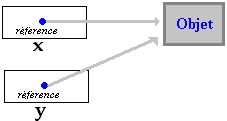


Deux objets Java seront instanciés dans la **machine virtuelle Java** de la manière suivante :



Attitude à rapprocher pour comparaison, à celle dont **Delphi** gère les objets dans une pile d'exécution de type LIFO et un tas :   
  
  Attention à l'utilisation de l'affectation entre variables d'objets dans le modèle de représentation par référence. L'affectation x = y ne recopie pas le bloc objet de données de y dans celui de x, mais seulement la référence (l'adresse) de y dans la référence de x. Visualisons cette remarque importante :

***Situation au départ, avant affectation***   


***Situation après l'affectation "  x = y "***   


En java, la désallocation étant automatique, le bloc de données objet qui était référencé par **y** avant l'affectation, n'est pas perdu, car le garbage collector  se charge de restituer la mémoire libérée au **segment de mémoire** de la **machine virtuelle Java**.   
    
*2.2 Les constructeurs d'objets*

Un constructeur est une **méthode spéciale** d'une classe dont la seule fonction est d'**instancier** un objet (créer le bloc de données). Comme en Delphi une **classe Java peut posséder plusieurs constructeur**s, il est possible de pratiquer des initialisations d'attributs dans un constructeur. Comme toutes les méthodes, un constructeur peut avoir ou ne pas avoir de paramètres formels.

* + Si vous ne déclarez pas de constructeur spécifique pour une classe, **par défaut** Java attribue  automatiquement un constructeur sans paramètres formels, portant le même nom que la classe. A la différence de Delphi où le nom du constructeur est quelconque, en Java le( ou les) **constructeur doit obligatoirement porter le même nom que la classe** (majuscules et minuscules comprises).
  + Un constructeur d'objet d'une classe n'a d'intérêt que s'il est visible par tous les programmes qui veulent instancier des objets de cette classe, c'est pourquoi l'on mettra toujours le mot clef **public** devant la déclaration du constructeur.
  + Un constructeur est une méthode spéciale dont la fonction est de créer des objets, dans son en-tête il n'a pas de type de retour et le mot clef **void** n'est pas non plus utilisé !

Soit une classe dénommée Un dans laquelle, comme nous l'avons fait jusqu'à présent nous n'indiquons aucun constructeur spécifique :

|  |
| --- |
| **class** **Un**   **{ int** a;   **}** |

Automatiquement Java attribue un constructeur public à cette classe **public** Un ( ). C'est comme si Java avait introduit dans votre classe à votre insu , une nouvelle méthode dénommée Un.Cette méthode "cachée" n'a aucun paramètre et aucune instruction dans son corps. Ci-dessous un exemple de programme Java correct illustrant ce qui se passe :

|  |
| --- |
| **class** **Un**   **{ public** Un ( ) **{** **}**  **int** a;   **}** |

|  |
| --- |
| * + **Vous pouvez programmer et personnaliser vos propres constructeurs.**   + **Une classe Java peut contenir plusieurs constructeurs dont les en-têtes diffèrent uniquement par la liste des paramètres formels.** |

***Exemple de constructeur avec instructions :***

|  |  |
| --- | --- |
| **Java** | **Explication** |
| **class** **Un**   **{ public** Un ( )   **{** a = 100  **}**  **int** a;   **}** | Le constructeur **public** Un sert ici à initialiser à 100 la valeur de l'attribut "**int** a" de chaque objet qui sera instancié. |

***Exemple de constructeur avec paramètre :***

|  |  |
| --- | --- |
| **Java** | **Explication** |
| **class** **Un**   **{ public** Un (**int** b )   **{** a = b;  **}**  **int** a;   **}** | Le constructeur **public** Un sert ici à initialiser la valeur de l'attribut "**int** a" de chaque objet qui sera instancié. Le paramètre **int** b contient cette valeur. |

***Exemple avec plusieurs constructeurs  :***

|  |  |
| --- | --- |
| **Java** | **Explication** |
| **class** **Un**   **{ public** Un (**int** b )   **{** a = b;  **}**  **public** Un ( )   **{** a = 100;  **}**  **public** Un (**float** b )   **{** a = (**int**)b;  **}**  **int** a;   **}** | La classe Un possède 3 constructeurs servant à initialiser chacun d'un manière différente le seul attribut **int** a. |

***Déclaration de constructeurs*** en ***Java***

|  |
| --- |
| **Java** |
| **class** **Un**   **{**  **public** Un ( )   **{** a = 100;  **}**  **public** Un (**int** b )   **{** a = b;  **}**  **public** Un (**float** b )   **{** a = (**int**)b;  **}**  **int** a;   **}** |

Tous les constructeurs doivent avoir le même nom .   
*2.3 Utilisation du constructeur d'objet automatique (par défaut)*

Le constructeur d'objet par défaut de toute classe Java comme nous l'avons signalé plus haut est une méthode spéciale sans paramètre, l'appel à cette méthode spéciale afin de construire un nouvel objet répond à une syntaxe spécifique par utilisation du mot clef **new**..

**Syntaxe**   
Pour un constructeur sans paramètres formels, l'instruction d'**instanciation d'un nouvel objet** à partir d'un identificateur de variable déclarée selon un type de classe, s'écrit syntaxiquement ainsi :   
constructor

|  |  |  |
| --- | --- | --- |
| ***Exemple*** : (deux façons équivalentes de créer un objet **x** de classe **Un**)   |  |  | | --- | --- | | **Un x** ;  **x** = **new Un**( ); | **Un x** = **new Un**( ); |   Cette instruction crée dans le segment de mémoire de la machine virtuelle Java, un nouvel objet de classe **Un** dont la référence (l'adresse) est mise dans la variable **x**. |

Dans l'exemple ci-dessous, nous utilisons le constructeur par défaut de la classe Un :   
**class** **Un**   
 **{** ...   
 **}**

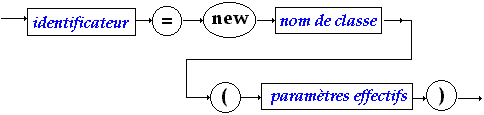
*// la déclaration :*   
**Un x , y ;**   
....   
*// la création :*   
**x** = **new Un**( );   
**y** = **new Un**( );

***Un programme de 2 classes, illustrant l'affectation de références :***

|  |  |
| --- | --- |
| **Java** | **Explication** |
| **class** AppliClassesReferences  **{**   **public static void** main(String [] arg) **{**    **Un** x,y ;    x = **new** **Un**( );    y = **new** **Un**( );    System.out.println("x.a="+x.a);    System.out.println("y.a="+y.a);    y = x;    x.a =12;    System.out.println("x.a="+x.a);    System.out.println("y.a="+y.a);    **}**   **}**   **class** **Un**   **{** **int** a=10;   **}** | Ce programme Java contient deux classes :  **class** AppliClassesReferences        et    **class** Un  La classe AppliClassesReferences est une classe exécutable car elle contient la méthode main. C'est donc cette méthode qui agira dès l'exécution du programme. |
| **Détaillons les instructions** | **Que se passe-t-il à l'exécution ?** |
| **Un** x,y ;    x = **new** **Un**( );    y = **new** **Un**( ); | Instanciation de 2 objets différents x et y de type **Un**. |
| System.out.println("x.a="+x.a);   System.out.println("y.a="+y.a); | *Affichage de :*  x.a = 10  y.a = 10 |
| y = x; | La référence de y est remplacée par celle de x dans la variable y (y pointe donc vers le même bloc que x). |
| x.a =12;   System.out.println("x.a="+x.a);   System.out.println("y.a="+y.a); | *On change la valeur de l'attribut a de x, et l'on demande d'afficher les attributs de x et de y* :  x.a = 12  y.a = 12  **Comme y pointe vers x, y et x sont maintenant le même objet sous deux noms différents !** |

*2.4 Utilisation d'un constructeur d'objet personnalisé*

L'utilisation d'un constructeur personnalisé d'une classe est semblable à celle du constructeur par défaut de la classe. La seule différence se trouve lors de l'instanciation : il faut fournir des paramètres effectifs lors de l'appel au constructeur.

**Syntaxe**   


***Exemple avec plusieurs constructeurs  :***

|  |  |
| --- | --- |
| **une classe Java** | **Des objets créés** |
| **class** **Un**   **{**  **int** a ;  **public** Un (**int** b ) **{**      a = b ;  **}**  **public** Un ( ) **{**      a = 100 ; **}**  **public** Un (**float** b ) **{**      a = (**int**)b ; **}**   **}** | **Un** obj1 = **newUn**( );  **Un** obj2 = **new** **Un**( 15 );  **int** k = 14;  **Un** obj3 = **new** **Un**( k );  **Un** obj4 = **new** **Un**( 3.25f );  **float** r = -5.6;  **Un** obj5 = **new** **Un**( r ); |

*2.5 Le mot clef this*

Il est possible de dénommer dans les instructions d'une méthode de classe, un futur objet qui sera instancié plus tard. Le paramètre ou (mot clef) **this** est implicitement présent dans chaque objet instancié et il contient la référence à l'objet actuel. Il joue exactement le même rôle que le mot clef **self** en Delphi.

|  |  |
| --- | --- |
| **Java** | **Java équivalent** |
| **class** **Un**   **{ public** Un ( )   **{** a = 100;  **}**  **int** a;   **}** | **class** **Un**   **{ public** Un ( )   **{** **this**.a = 100;  **}**  **int** a;   **}** |

Dans le programme de droite le mot clef this fait référence à l'objet lui-même, ce qui dans ce cas est superflu puisque la variable **int** a est un champ de l'objet.

Montrons deux exemples d'utilisation pratique de this

***Cas où l'objet est passé comme un paramètre dans une de ses méthodes :***

|  |  |
| --- | --- |
| **Java** | **Explications** |
| **class** **Un**   **{ public** Un ( )   **{** a = 100;  **}**  **public void** methode1(Un x)    **{**    System.out.println("champ a ="+x.a);    **}**    **public void** methode2( **int** b )    **{**  a += b;        methode1(**this**);    **}**  **int** a;   **}** | La methode1(Un x) reçoit un objet de type Exemple en paramètre et imprime son champ **int** a.  La methode2( **int** b ) reçoit un entier **int** b qu'elle additionne au champ **int** a de l'objet, puis elle appelle la méthode1 avec comme paramètre l'objet lui-même. |

***Cas où le this sert à outrepasser le masquage de visibilité :***

|  |  |
| --- | --- |
| **Java** | **Explications** |
| **class** **Un**   **{**  **int** a;  **public void** methode1(**float** a)     **{**    a = **this**.a + 7 ;     **}**  **}** | La methode1(**float** a) possède un paramètre **float** a dont le nom masque le nom du champ **int** a.  Si nous voulons malgré tout accéder au champ de l'objet, l'objet étant référencé par **this**, "**this**.a" est donc le champ **int** a de l'objet lui-même. |

***Comparaison Delphi - java sur ce second exemple (similitude complète aussi)***

|  |  |
| --- | --- |
| **Delphi** | **Java** |
| **Un** = **class**      a : integer;    **public**     **procedure** methode( a:real );  **end**;  **implementation**  **procedure** **Un**.methode( a:real );**begin**   a = **self**.a + 7 ;  **end**; | **class** **Un**   **{**  **int** a;  **public void** methode(**float** a)     **{**    a = **this**.a + 7 ;     **}**  **}** |

**3. Variables et méthodes**

Nous examinons dans ce paragraphe comment Java utilise les variables et les méthodes à l'intérieur d'une classe. Il est possible de modifier des variables et des méthodes d'une classe ceci sera examinée plus loin.

En Java, les champs et les méthodes sont classés en deux catégories :

* + **Variables et méthodes de classe**
  + **Variables et méthodes d'instance**

*3.1 Variables dans une classe en général*

Rappelons qu'en Java, nous pouvons déclarer dans un bloc (for, try,...) de nouvelles variables à la condition qu'elles n'existent pas déjà dans le corps de la méthode où elles sont déclarées. Nous les dénommerons : **variables locales de méthode**.

***Exemple de variables locales de méthode :***

|  |  |
| --- | --- |
| **class** Exemple  **{**  **void** calcul ( **int** x, **int** y )  **{int** a = 100;      **for** ( **int** i = 1; i<10; i++ )      **{char** carlu;         System.out.print("Entrez un caractère : ");          carlu = Readln.unchar( );          **int** b =15;         a =....            .....       **}**  **}**  **}** | La définition **int** a = 100; est locale à la méthode en général  La définition **int** i = 1; est locale à la boucle **for**.  Les définitions **char** carlu et **int** b sont locales au corps de la boucle **for**. |

Java ne connaît pas la notion de variable globale au sens habituel donné à cette dénomination, dans la mesure où toute variable ne peut être définie qu'à l'intérieur d'une classe, ou d'une méthode inclue dans une classe. Donc mis à part les **variables locales de méthode** définies dans une méthode, Java reconnaît une autre catégorie de variables, ***les variables définies dans une classe mais pas à l'intérieur d'une méthode spécifique***. Nous les dénommerons  : **variables de classes**.

***Exemple de variables de classe :***

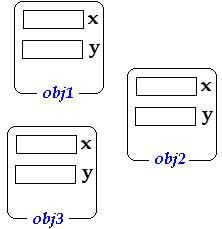
|  |  |  |
| --- | --- | --- |
| **class** AppliVariableClasse  **{**  **float** r ;  **void** calcul ( **int** x, **int** y )  **{** ....  **}**  **int** x =100;  **int** valeur ( **char** x )  **{** .....  **}**  **long** y;  **}** | Les variables **float** r , **long** y et **int** x sont des variables de classe.   |  | | --- | | La position de la déclaration de ces variables n'a aucune importance. Elles sont **visibles dans tout le bloc classe** (c'est à dire visibles par toutes les méthodes de la classe). |   **Conseil** : regroupez les variables de classe au début de la classe afin de mieux les gérer. |

*3.2 Variables et méthodes d'instance*

Java se comporte comme un langage orienté objet classique vis à vis de ses variables et de ses méthodes.  A chaque instanciation d'un nouvel objet d'une classe donnée, la machine virtuelle Java enregistre le p-code des méthodes de la classe dans la **zone de stockage** des méthodes,elle alloue dans le **segment de mémoire** autant d'emplacements mémoire pour les variables que d'objet créés. Java dénomme cette catégorie **les variables et les méthodes d'instance**.

|  |  |
| --- | --- |
| **une classeJava** | **Instanciation de 3 objets** |
| **class** AppliInstance  **{** **int** x ;     **int** y ;  **}** | AppliInstance  obj1 = **new** AppliInstance( );  AppliInstance  obj2 = **new**AppliInstance( );  AppliInstance  obj3 = **new**AppliInstance( ); |

***Voici une image du segment de mémoire associé à ces 3 objets :***



***Un programme Java à 2 classes illustrant l'exemple précédent :***

|  |
| --- |
| **Programme Java exécutable** |
| **class** AppliInstance  **{**  **int** x = -58 ;     **int** y = 20 ;  **}**  **class** Utilise  **{** **public static void** main(String [ ] arg) **{**       AppliInstance  obj1 = **new** AppliInstance( );       AppliInstance  obj2 = **new** AppliInstance( );       AppliInstance  obj3 = **new** AppliInstance( );       System.out.println( "obj1.x = " + obj1.x );    **}**  **}** |

*3.3 Variables et méthodes de classe - static*

On identifie une variable ou une méthode de classe en précédant sa déclaration du mot clef **static**. Nous avons déjà pris la majorité de nos exemples simples avec de tels composants.

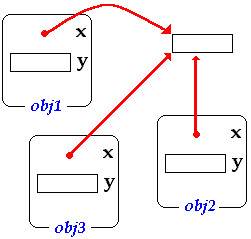
Voici deux déclaration de variables de classe :

**static int** x ;   
**static int** a = 5;

Une variable de classe est accessible (selon sa visibilité) sans avoir à instancier un objet de la classe, mais uniquement en référençant la variable par le nom de la classe dans la notation de chemin uniforme d'objet.

|  |  |
| --- | --- |
| **une classeJava** | **Instanciation de 3 objets** |
| **class** AppliInstance  **{** **static int** x ;     **int** y ;  **}** | AppliInstance  obj1 = **new** AppliInstance( );  AppliInstance  obj2 = **new**AppliInstance( );  AppliInstance  obj3 = **new**AppliInstance( ); |

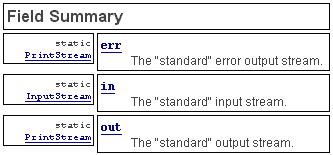
***Voici une image du segment de mémoire associé à ces 3 objets :***



***Exemple de variables de classe :***

|  |  |
| --- | --- |
| **class** ApplistaticVar  **{** **static int** x =15 ;  **}**  **class** UtiliseApplistaticVar  **{**  **int** a ;     **void** f( )    **{** a = ApplistaticVar.x ;      .....     **}**  **}** | La définition "**static int** x =15 ;" crée une variable de la classe ApplistaticVar, nommée x.  L'instruction "a = ApplistaticVar.x ;" utilise la variable x comme variable de classe ApplistaticVar sans avoir instancié un objet de cette classe. |

Nous utilisons sans le savoir depuis le début de ce cours, une variable de classe sans jamais instancier un quelconque objet de la classe. Dans l'instruction << **System.out.println("Bonjour");** >>, la classe **System** possède une variable (un champ)de classe **out** qui est elle-même un objet de classe dérivant de la classe **FilterOutputStream**, nous n'avons jamais instancié d'objet de cette classe **System**.

***Les champs de la classe System :***   


Notons que les champs **err** et **in** sont aussi des variables de classe (précédées par le mot **static**). Résumons ci-dessous ce qu'il faut connaître pour bien utiliser ces outils.

Une méthode de classe est une méthode dont l'implémentation est la même pour tous les objets de la classe, en fait la différence avec une méthode d'instance a lieu sur la catégorie des variables sur lesquelles ces méthodes agissent.

**De par leur définition les méthodes de classe ne peuvent travailler qu'avec des variables de classe, alors que les méthodes d'instances peuvent utiliser les deux catégories de variables.**

***Un programme correct illustrant le discours :***

|  |  |
| --- | --- |
| **Java** | **Explications** |
| **class** Exemple  **{static int** x  ;    **int** y  ;   **void** f1(**int** a)    **{** x = a;      y = a;     **}**   **static void** g1(int a)    **{** x = a;    **}**  **}**  **class** Utilise  **{**   **public static void** main(String [] arg)    **{** Exemple obj = **new** Exemple( );    obj.f1(10);    System.out.println("<f1(10)>obj.x="+obj.x);    obj.g1(50);    System.out.println("<g1(50)>obj.x="+obj.x);   **}**  **}** | **void** f1(**int** a)    **{** x = a; *//accès à la variable de classe*       y = a ;  *//accès à la variable d'instance*    **}**  **static void** g1(**int** a)    **{**  x = a;  *//accès à la variable de classe*       y = a ; *//engendrerait un erreur de compilation : accès à une variable non static interdit !*    **}**  La méthode f1 accède à toutes les variables de la classe Exemple, la méthode g1 n'accède qu'aux variables de classe (**static**).  ***Après exécution on obtient :***  <f1(10)>obj.x = 10  <g1(50)>obj.x = 50 |

**Bilan pratique et utile sur les composants de classe, en 5 remarques**

**1)** - Les méthodes et les variables de classe sont **précédées obligatoirement** du mot clef **static**.

Exemple :

|  |  |
| --- | --- |
| **Java** | **Explications** |
| **class** **Exemple1**   **{**  **int** a = 5;  **static int** b = 19;     **void** m1( ){...}     **static void** m2( ) {...}   **}** | La variable a dans **int** a = 5; est une variable d'instance.  La variable b dans **static int** b = 19; est une variable de classe.  La méthode m2 dans **static void** m2( ) {...} est une méthode de classe. |

**2)** - Pour utiliser une variable **x1** ou une méthode **meth1** de la classe **Classe1**, il suffit de d'écrire **Classe1.x1** ou bien **Classe1.meth1**.

Exemple :

|  |  |
| --- | --- |
| **Java** | **Explications** |
| **class** **Exemple2**   **{**  **static int** b = 19;     **static void** m2( ) {...}   **}**  **class** UtiliseExemple  **{** **Exemple2**.b = 53;    **Exemple2**.m2( );    ...  **}** | Dans la classe Exemple2 b est une variable de classe, m2 une méthode de classe.  La classe UtiliseExemple fait appel à la méthode m2 directement avec le nom de la classe, il en est de même avec le champ b de la classe Exemple2  . |

**3)** - Une variable de classe (précédée du mot clef **static**) est **partagée par tous les objets** de la même classe.

Exemple :

|  |  |
| --- | --- |
| **Java** | **Explications** |
| **class** AppliStatic  **{** **static int** x = -58 ;     **int** y = 20 ;     ...  **}**  **class** Utilise  **{**  **public static void** main(String [] arg) **{**     AppliStatic  obj1 = **new** AppliStatic( );     AppliStatic  obj2 = **new** AppliStatic( );     AppliStatic  obj3 = **new** AppliStatic( );     obj1.y = 100;     obj1.x = 101;     System.out.println("obj1.x="+obj1.x);     System.out.println("obj1.y="+obj1.y);     System.out.println("obj2.x="+obj2.x);     System.out.println("obj2.y="+obj2.y);     System.out.println("obj3.x="+obj3.x);     System.out.println("obj3.y="+obj3.y);     AppliStatic.x = 99;     System.out.println(AppliStatic.x="+obj1.x);   **}**  **}** | Dans la classe AppliStatic **x** est une variable de classe, et **y** une variable d'instance.  La classe Utilise crée 3 objets (obj1,obj2,obj3) de classe AppliStatic.  L'instruction obj1.**y** = 100; est un accès au champ **y** de l'instance obj1. Ce n'est que le champ **x** de cet objet qui est modifié,les champs **x** des objets obj2 et obj3 restent inchangés  Il y a deux manières d'accéder à la variable static **x**, :   * + soit comme un champ de l'objet (accès semblable à celui de **y**) : obj1.**x** = 101;   + soit comme une variable de classe proprement dite :  AppliStatic.**x** = 99;   Dans les deux cas cette variable **x** est modifiée globalement et donc tous les champs **x** des 2 autres objets, obj2 et obj3 prennent la nouvelle valeur. |

Au début lors de la création des 3 objets chacun des champs x vaut -58 et des champs y vaut 20, l'affichage par System.out.println(...) donne les résultats suivants qui démontrent le partage de la variable x par tous les objets.

***Après exécution :***   
obj1.x = 101   
obj1.y = 100   
obj2.x = 101   
obj2.y = 20   
obj3.x = 101   
obj3.y = 20   
<AppliStatic>obj1.x = 99

**4)** - Une méthode de classe (précédée du mot clef **static**) **ne peut utiliser que des variables de classe** (précédées du mot clef **static**) et jamais des variables d'instance.Une méthode d'instance peut accéder aux deux catégories de variables

 **5)** - Une méthode de classe (précédée du mot clef **static**) **ne peut  appeller** (invoquer) **que des méthodes de classe** (précédées du mot clef **static**).

Exemple :

|  |  |
| --- | --- |
| **Java** | **Explications** |
| **class** AppliStatic  **{**  **static int** x = -58 ;     **int** y = 20 ;  **void** f1(**int** a)    **{** AppliStatic.x = a;       y = 6 ;    **}**  **}**  **class** Utilise  **{**  **static void** f2(int a)     **{** AppliStatic.x = a;     **}**  **public static void** main(String [] arg) **{**     AppliStatic  obj1 = new AppliStatic( );     AppliStatic  obj2 = new AppliStatic( );     AppliStatic  obj3 = new AppliStatic( );     obj1.y = 100;     obj1.x = 101;     AppliStatic.x = 99;      f2(101);     obj1.f1(102);   **}**  **}** | Nous reprenons l'exemple précédent en ajoutant à la classe AppliStatic une méthode interne f1 :  **void** f1(**int** a)  **{**      AppliStatic.x = a;      y = 6 ;    **}**  Cette méthode accède à la variable de classe comme un champ d'objet.  Nous rajoutons à la classe Utilise, un méthode static (méthode de classe) notée f2:  **static void** f2(int a)     **{** AppliStatic.x = a;     **}**  Cette méthode accède elle aussi à la variable de classe parce qu c'est une méthode static.  Nous avons donc **quatre** manières d'accéder à la variable static **x**, :   * + soit comme un champ de l'objet (accès semblable à celui de **y**) : obj1.**x** = 101;   + soit comme une variable de classe proprement dite :  AppliStatic.**x** = 99;   + soit par une méthode d'instance sur son champ : obj1.f1(102);   + soit par une méthode static (de classe) : f2(101); |

Comme la méthode main est static, elle peut invoquer la méthode f2 qui est aussi statique.

Au paragraphe précédent, nous avons indiqué que Java ne connaissait pas la notion de variable globale stricto sensu, mais en fait une variable **static peut jouer le rôle d'un variable globale pour un ensemble d'objets** instanciés à partir de la même classe.   
    
*3.4 Surcharge et polymorphisme*

***Vocabulaire :***   
Le polymorphisme est la capacité d'une entité à posséder plusieurs formes. En informatique ce vocable s'applique aux méthodes selon leur degré d'adaptabilité,  nous distinguons alors deux dénomination :

|  |
| --- |
| * + le polymorphisme statique ou **la surcharge de méthode**   + le polymorphisme dynamique ou **la redéfinition de méthode** ou encore la **surcharge héritée**. |

A- La surcharge de méthode (polymorphisme statique) est une fonctionnalité classique des langages très évolués et en particulier des langages orientés objet; elle consiste dans le fait qu'une classe peut disposer de **plusieurs méthodes ayant le même nom**, mais avec des paramètres formels différents ou éventuellement un type de retour différent. On appelle **signature** de la méthode l'en-tête  de la méthode avec ses paramètres formels. Nous avons déjà utilisé cette fonctionnalité précédement dans le paragraphe sur les constructeurs, où la classe **Un** disposait de trois constructeurs surchargés :   
**class** **Un**   
 **{**   
**int** a;   
**public** Un ( )   
**{** a = 100;**}**

**public** Un (**int** b )   
**{** a = b;**}**

**public** Un (**float** b )   
**{** a = (**int**)b;**}**   
 **}**

**Mais cette surcharge est possible aussi pour n'importe quelle méthode de la classe autre que le constructeur.** Le compilateur n'éprouve aucune difficulté lorsqu'il rencontre un appel à l'une des versions surchargée d'une méthode, il cherche dans la déclaration de toutes les surcharges celle dont la **signature** (la déclaration des paramètres formels) coïncide avec les paramètres effectifs de l'appel.

|  |  |
| --- | --- |
| **Programme Java exécutable** | **Explications** |
| **class** Un  **{** int a;    **public** Un (**int** b )     { a = b; }    **void** f ( )    **{** a \*=10;  **}**  **void** f ( **int** x )    **{** a +=10\*x; **}**  **int** f ( **int** x, **char** y )  **{** a = x+(**int**)y;  **return** a;  **}**  **}**  **class** AppliSurcharge  **{**     **public static void** main(String [ ] arg) **{**     Un obj = **new** Un(15);     System.out.println("<création> a ="+obj.a);     obj.f( );     System.out.println("<obj.f()> a ="+obj.a);     obj.f(2);     System.out.println("<obj.f()> a ="+obj.a);     obj.f(50,'a');     System.out.println("<obj.f()> a ="+obj.a);     **}**  **}** | La méthode **f** de la classe **Un** est surchargée trois fois :  **void** f ( )    **{** a \*=10;  **}**  **void** f ( **int** x )    **{** a +=10\*x; **}**  **int** f ( **int** x, **char** y )  **{** a = x+(int)y;  **return** a;  **}**  La méthode f de la classe Un peut donc être appelée par un objet instancié de cette classe sous l'une quelconque des trois formes :  obj.f( ); pas de paramètre => choix : **void** f ( )  obj.f(2); paramètre **int** => choix : **void** f ( **int** x )  obj.f(50,'a'); deux paramètres, un int un char => choix : **int** f ( **int** x, **char** y ) |

B -  la redéfinition de méthode (ou polymorphisme dynamique) est spécifique aux langages orientés objet. Elle est mise en oeuvre lors de l'héritage d'une classe mère vers une classe fille dans le cas d'une méthode ayant la même signature dans les deux classes. Dans ce cas les actions dûes à l'appel de la méthode, dépendent du code inhérent à chaque version de la méthode (celle de la classe mère, ou bien celle de la classe fille). Ces actions peuvent être différentes. En java aucun mot clef n'est nécessaire ni pour la surcharge ni pour la redéfinition, c'est le compilateur qui analyse la syntaxe afin de de se rendre compte en fonction des signatures s'il s'agit de redéfinition ou de surcharge. Attention il n'en va pas de même qu'en Delphi, plus lourd mais plus explicite pour le programmeur, qui nécessite des mots clefs comme virtual, dynamic override et overload.

Dans l'exemple ci-dessous la classe ClasseFille qui hérite de la classe ClasseMere, redéfini la méthode **f** de sa classe mère :

Comme delphi, Java peut combiner la surcharge et la redéfinition sur une même méthode, c'est pourquoi nous pouvons parler de surcharge héritée :

|  |
| --- |
| **Java** |
| **class** ClasseMere  **{**    **int** x = 10;  **void** f ( **int** a)    **{** x +=a; **}**    **void** g ( **int** a, **int** b)     **{** x +=a\*b; **}**  **}**  **class** ClasseFille **extends** ClasseMere  **{**     **int** y = 20;    **void** f ( **int** a) *//redéfinition*    **{** x +=a; **}**  **void** g (**char** b) *//surcharge et redéfinition de g*    **{** ...... **}**  **}** |

C'est le compilateur Java qui fait tout le travail. Prenons un objet obj de classe Classe1, lorsque le compilateur Java trouve une instruction du genre "obj.**method1**(paramètres effectifs);", sa démarche d'analyse est sembleble à celle du compilateur Delphi, il cherche dans l'ordre suivant :

* + Y-a-t-il dans Classe1, une méthode qui se nomme **method1** ayant une signature identique aux paramètres effectifs ?
  + si oui c'est la méthode ayant cette signature qui est appelée,
  + si non le compilateur remonte dans la hierarchie des classes mères de Classe1 en posant la même question récursivement jusqu'à ce qu'il termine sur la classe Object.
  + Si aucune méthode ayant cette signature n'est trouvée il signale une erreur.

Soit à partir de l'exemple l'exemple précédent  les instructions suivantes :   
ClasseFille obj = **new** ClasseFille( );   
obj.**g**(-3,8);   
obj.**g**('h');

Le compilateur Java applique la démarche d'analyse décrite, à l'instruction "obj.**g**(-3,8);". Ne trouvant pas dans ClasseFille de méthode ayant la bonne signature (signature = deux entiers) , le compilateur remonte dans la classe mère ClasseMere et trouve une méthode " **void** g ( **int** a, **int** b) " de la classe ClasseMere ayant la bonne signature (signature = deux entiers), il procède alors à l'appel de cette méthode sur les paramètres effectifs (-3,8).

Dans le cas de l'instruction obj.**g**('h'); , le compilateur trouve immédiatement dans ClasseFille la méthode " **void** g (**char** b) " ayant la bonne signature, c'est donc elle qui est appelée sur le paramètre effectif 'h'.   
 

**Résumé pratique sur le polymorphisme en Java**

|  |
| --- |
| La **surcharge** (polymorphisme statique) consiste à proposer différentes signatures de la même méthode. |
| La **redéfinition** (polymorphisme dynamique) ne se produit que dans l'héritage d'une classe par redéfinition de la méthode mère avec une méthode fille (ayant ou n'ayant pas la même signature). |

*3.5 Le mot clef* ***super***

Nous venons de voir que le compilateur s'arrête dès qu'il trouve une méthode ayant la bonne signature dans la hiérarchie des classes, il est des cas où voudrions accéder à une méthode de la classe mère alors que celle-ci est redéfinie dans la classe fille. C'est un problème analogue à l'utilisation du this lors du masquage d'un attribut. Il existe un mot clef qui permet d'accéder à la classe mère (classe immédiatement au dessus): le mot **super**.

On parle aussi de super-classe au lieu de classe mère en Java. Ce mot clef **super** référence la classe mère et à travers lui, il est possible d'accéder à tous les champs et à toutes les méthodes de la super-classe (classe mère). Ce mot clef est très semblable au mot clef **inherited** de Delphi qui joue le même rôle uniquement sur les méthodes.

***Exemple :***

|  |
| --- |
| **Java** |
| **class** ClasseMere  **{**    **int** x = 10;  **void** g ( **int** a, **int** b)     **{** x +=a\*b; **}**  **}**  **class** ClasseFille **extends** ClasseMere  **{**     **int** x = 20; *//masque le champ x de la classe mère*  **void** g (**char** b) *//surcharge et redéfinition de g*    **{**  super.x = 21; *//accès au champ* ***x*** *de la classe mère*    super.g(-8,9); *//accès à la méthode* ***g*** *de la classe mère*  **}**  **}** |

Le mot clef super peut en Java être utilisé seul ou avec des paramètres comme un appel au constructeur de la classe mère.

***Exemple :***

|  |
| --- |
| **Java** |
| **class** ClasseMere  **{**    **public** ClasseMere ( ) **{**    ... **}**  **public** ClasseMere (**int** a ) **{**    ... **}**  **}**  **class** ClasseFille **extends** ClasseMere  **{**     **public** ClasseFille ( ) **{**     **super** ( ); *//appel au 1er constructeur de ClasseMere*  **super** ( 34 ); *//appel au 2nd constructeur de ClasseMere*      ... **}**  **public** ClasseFille ( **char** k, **int** x) **{**  **super** ( x ); *//appel au 2nd constructeur de ClasseMere*      ... **}**  **}** |

*3.6 Modification de visibilité*

Terminons ce chapitre par les classiques modificateurs de visibilité des variables et des méthodes dans les langages orientés objets, dont Java dispose :

***Modification de visibilité (modualrité public-privé)***

|  |  |
| --- | --- |
| **par défaut (aucun mot clef)** | les variables et les méthodes d'une classe non précédées d'un mot clef sont visibles par toutes les classes inclues dans le module seulement. |
| **public** | les variables et les méthodes d'une classe  précédées du mot clef **public** sont visibles par toutes les classes de tous les modules. |
| **private** | les variables et les méthodes d'une classe  précédées du mot clef **private** ne sont visibles que dans la classe seulement. |
| **protected** | les variables et les méthodes d'une classe  précédées du mot clef **protected** sont visibles par toutes les classes inclues dans le module, et par les classes dérivées de cette classe. |

Ces attributs de visibilité sont identiques à ceux de Delphi.