Université Mohammed V
Faculté des Sciences, Rabat
Département d'Informatique

PROGRAMMATION ORIENTEE OBJET SOUS JAVA

Licence Fondamentale : Sciences Mathématiques Appliquées

Préparer par :
Pr. Soumia ZITI
Pr. Fouzia OMARY

Année universitaire 2011 / 2012

Présentation générale du langage Java

Historique :

Java signifie café en Slang (argot américain). Le langage Java est un langage de programmation informatique orienté objet créé par James Gosling et Patrick Naughton, employés de Sun Microsystems, avec le soutien de Bill Joy (cofondateur de Sun Microsystems en 1982), présenté officiellement le 23 mai 1995 au SunWorld. Le but était d'avoir un langage de développement simple et portable sur plusieurs systèmes d'exploitation tels que UNIX, Windows, Mac OS ou GNU/Linux, avec ou sans modifications, afin de programmer tous les processeurs (ordinateurs ou appareils électroménagers, …) tout en veillant à la robustesse, la compatibilité, la petite taille du runtime ou des codes générés et aussi facilité de programmation. Java reprend la syntaxe de C++ tout en le simplifiant et offre aussi un ensemble de classes pour développer des applications de types très variés (réseau, interface graphique, multi-tâches, etc.). Java comprend bien d'autres aspects (programmation graphique, applets, programmation réseau, multi-tâches)

Le langage Java a connu plusieurs évolutions depuis le JDK (Java Development Kit) 1.0 A partir de la version Java 1.2 on parle de Java 2 avec 2 grandes inovations : l'API graphique Swing est intégrée ; et la machine virtuelle Java de Sun inclut un compilateur "Juste à temps" (Just in Time).

L'environnement Java et le JDK

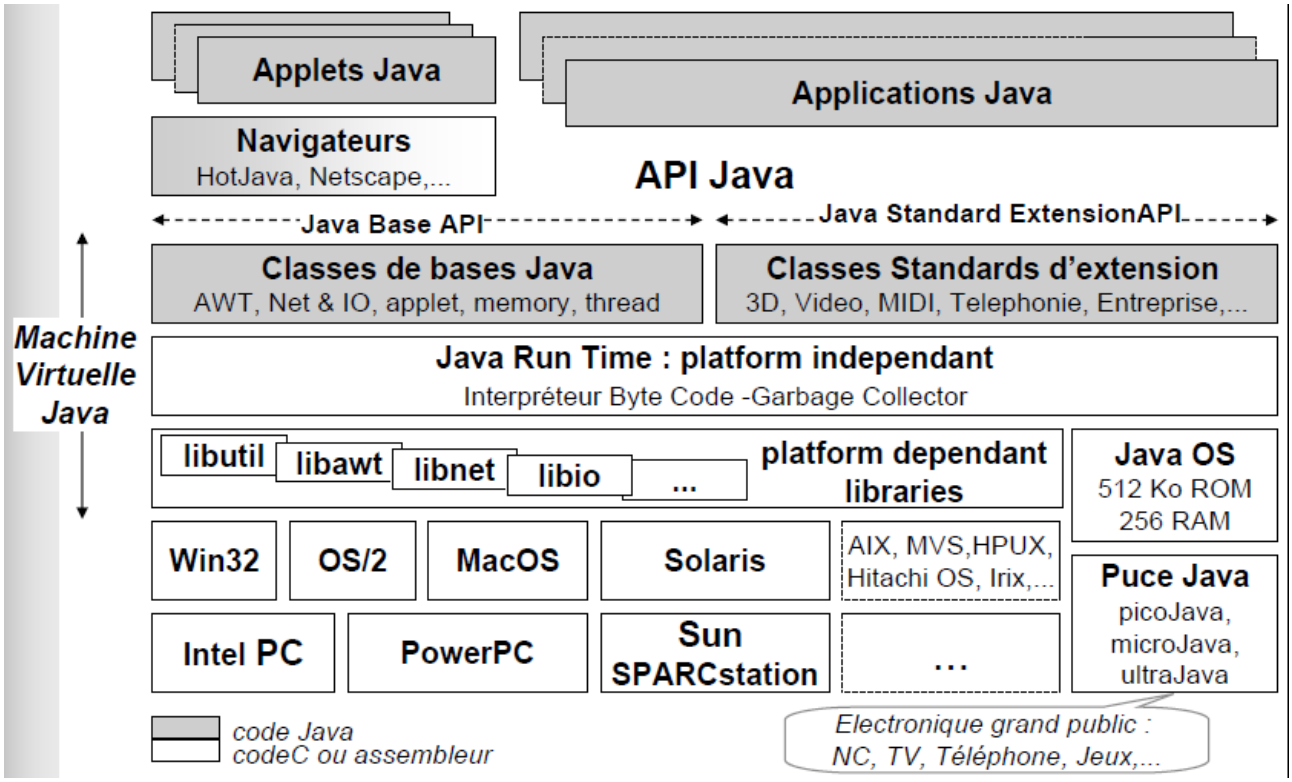
Java est situé entre SMALLTALK et C++, permet de développer des applications de taille importante intégrant les besoins de l'informatique actuelle et répondant à des objectifs de portabilité en utilisant une plateforme de déploiement basée sur une machine virtuelle (JVM) et des librairies de base (API) et aussi un ensemble d'outils qui le JDK

Le Java Development Kit (JDK) désigne un ensemble de bibliothèque logicielle de base du langage de programmation Java, ainsi que les outils avec lesquels le code Java peut être compilé, transformé en bytecode destiné à la machine virtuelle Java.

Il existe plusieurs éditions de JDK, selon la plate-forme Java1 considérée (et bien évidemment la version de Java ciblée) :

- JSE pour la Java 2 Standatd Edition également désignée J2SE ;
- JEE, sigle de Java Enterprise Edition également désignée J2EE ;
- JME 'Micro Edition', destinée au marché mobiles;
- etc.

À chacune de ces plateformes correspond une base commune de Development Kits, plus des bibliothèques additionnelles spécifiques selon la plate-forme Java que le JDK cible, mais le terme de JDK est appliqué indistinctement à n'importe laquelle de ces plates-formes.



Caractéristiques du langage Java

Java est un langage :

Simple :

la syntaxe est proche du langage C++ et C, mais sans utiliser les pointeurs. Le code source est organisé dans des packages avec des régles d'accés avec une gestion explicite de la mémoire (garbage collector). Java gère aussi bien objets que les types primitifs (qui peuvent aussi être des objets), les objets sont définies en utilisant le concepte des classe. Contrairement à C++, il ne permet pas l'héritage multiple, mais il offre la possibilité d'utiliser des interfaces tout en utilisant une librairie de classes (sockets, BD, graphiques...)

Orienté Objet :

ce paradigme consiste à associer au même endroit (l'objet) les différents types de données ou attributs et les différentes opérations qui peuvent manipuler ces données ce qui rend le code clair, rapide et réutilisable. En Java, tout se trouve dans une classe, il ne peut y avoir de déclarations ou de code en dehors du corps d'une classe

Interprété :

En Java, la compilation ne traduit pas directement le programme source dans le code natif de l'ordinateur. Le code source est d'abord traduit dans un langage binaire intermédiaire appelé "bytecode",qui est un langage d'une machine virtuelle (JVM – Java Virtual Machine) définie par Sun. Ce bytecode, ne dépend pas de l'environnement de travail où le code source est compilé. Au moment de l'exécution, il sera traduit dans le langage machine relatif à la machine sur laquelle il sera exécuté

Portable :

Un programmes écrits en Java fonctionnent de manière parfaitement similaire sur différentes architectures matérielles. On peut dès lors effectuer le développement sur une architecture donnée et faire tourner l'application sur toutes les autres quelles que soient les interfaces (Windows,Linux...) et

les versions (Windows 95,98,Me...).

distribué :

Java propose une API réseau standard qui permet de manipuler, par exemple, les protocoles HTTP & FTP avec aisance et aussi des API pour la communication entre des objets distribués (Remote Method Invocation).

Sécurisé :

La plate-forme Java fut l'un des premiers systèmes garantissant une sérieuse Exécution sécurisée de code distant cependant cette sécurité en vérifiant toujours le bytecode, en utilisant un chargeur de classe (classe Loader) ou en protégeant les fichiers ou les accès réseau. Le même modèle de sécurité est appliqué pour les application et pour les applets, locales ou téléchargées.

L'environnement de génération et d'exécution

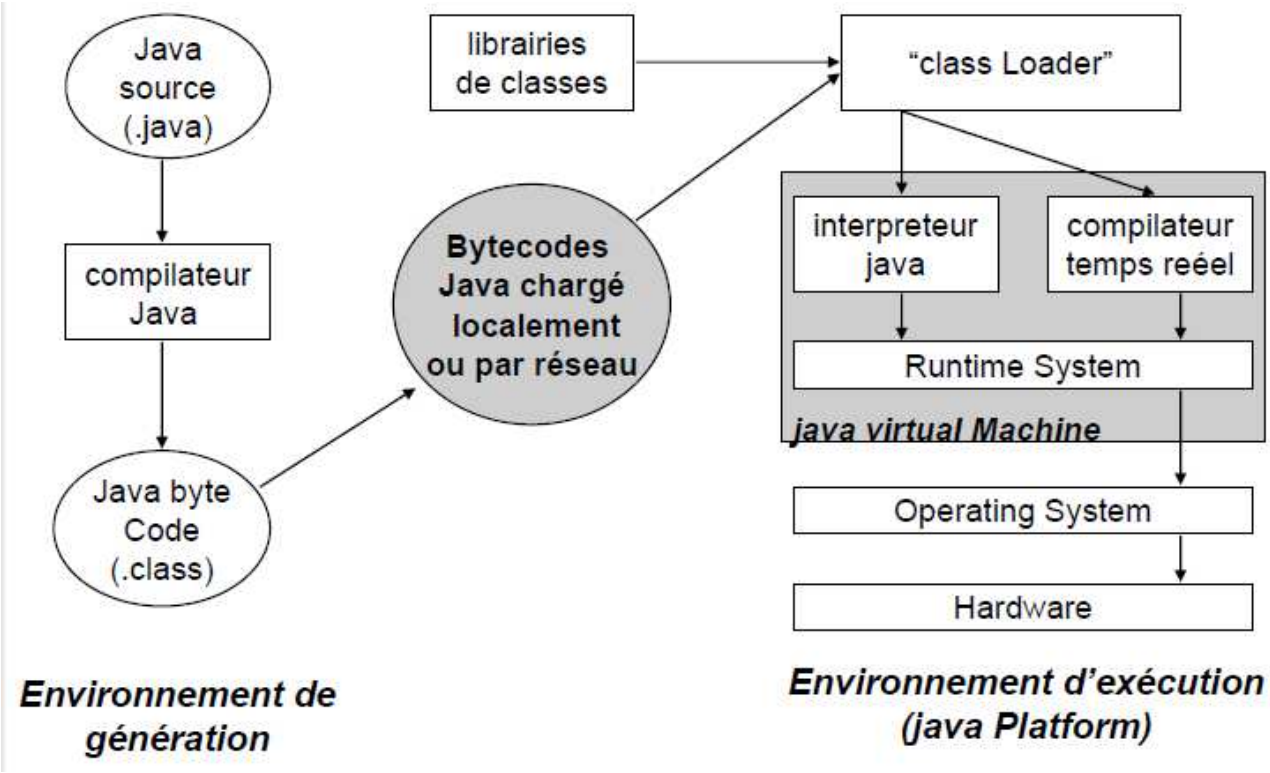


Plate-forme JAVA et librairies (API)

Elles diminuent la charge de travail, standardisent les applications, et fournissent des exemples de source Java de référence

Librairies standards :

java.lang : Types de bases, Threads, ClassLoader, Exception, Math, ...

java.util : Collections (Hashmap,ArrayList,TreeMap…), Ressources, Logging, Compression, Préférences

java.applet : des programmes Java utilisables sur le web avec des fonctionnalités interactives

java.awt,javax.swing : Interfaces Graphiques

java.io: Accès aux I/O par flux

java.net: Socket (UDP, TCP, multicast), URL, ...

java.lang.reflect : Introspection

java.beans : Composants logiciels

java.sql,javax.sql: Accès aux bases de données

java.security : signature, cryptographie, authentification

java.rmi : Remote Method Invocation

Premier programme sous java

Considérons la classe 'HelloWorld' définie ci-desous:

```
public class HelloWorld {
     public static void main(String args[]) {
          System.out.println(" Bonjour tout le monde " );
     }
}
```

- Ce code doit être sauvegardé obligatoirement dans un fichier texte (fichier source) dont le nom est le nom de la classe avec l'extension ' .java ' donc le fichier source sera nommé HelloWorld**.java'**
- Cette classe est exécutable car elle posséder la méthode '*main()'* ayant la signature :

```
public static void main(String[] args).
```

Pour compiler, on utilise la commande ***javac***: **javac HelloWorld.java** ce qui génère un fichier nommée « HelloWorld.class » appelé le bytecode.

Pour l'exécution, on utilise la commande ***java***: **java HelloWorld.** La machine virtuelle JVM interprète le *bytecode* de la méthode « *main() »* de la classe HelloWorld Q Aet affiche à l'écran la chaîne de caractères: Bonjour tout le monde

De manière générale tout programme destiné à être exécuté, doit contenir une méthode particulière nommée '`main()`'. Elle contient le «programme principal» à exécuter. Elle est définie de la manière suivante:

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

Le paramètre `args`, exigé par le compilateur Java, est un tableau d'objets de type `String`.

- La méthode ' main()' ne peut pas retourner d'entier comme en C.
- La classe contenant la méthode `main()` doit obligatoirement être `public` afin que la machine virtuelle y accède.
- Dans l'exemple précédent, le contenu de la classe « HelloWorld » est réduit à la définition d'une méthode `main()`.
- Un fichier source peut contenir plusieurs classes mais une seule doit être public
- Le nom du fichier source est identique au nom de la classe publique qu'il contient avec l'extension '.java'

Structure du langage Java

Types de données élémentaires

En Java, toute donnée appartient à un type bien précis. Il existe des données dites de *type primitif* et des données de *type objet*.

Il existe huit types primitifs en Java : quatre variantes de nombres entiers, deux variantes de nombres flottants (à virgule), un type booléen (vrai ou faux) et un type caractère. Un littéral est la représentation dans le code source d'une valeur de type primitif.

Entiers et flottants

Il existe, en Java, deux sortes de valeurs numériques : *les entiers*, qui n'ont pas de partie fractionnaire, et les *flottants*, qui en ont. Les flottants sont parfois appelés nombres à virgule flottante ou réels

Il y a quatre sortes de données entières (`byte`, `short`, `int` et `long`) et deux sortes de flottants (`float` et `double`). Tous ces types diffèrent par la quantité d'espace mémoire nécessaire pour stocker une valeur de ce type, ce qui détermine l'ensemble des valeurs que peuvent prendre chaque type. La taille de chacun de ces types est la même pour toutes les plateformes et est reprise dans le tableau suivant avec les valeurs minimales et maximales.

Type	Taille	Minimum	Maximum
`byte`	8 bits	-128	127
`short`	16 bits	-32 768	32 767
`int`	32 bits	-2 147 483 648	2 147 483 647
`long`	64 bits	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
`float`	32 bits	-3.4e+38 (7 chiffres significatifs)	3.4e+38 (7 chiffres significatifs)
`double`	64 bits	-1.7e+308 (15 chiffres significatifs)	1.7e+308 (15 chiffres significatifs)

Littéraux numériques

Les entiers simplement en donnant leur valeur sous forme décimale, octale ou hexadécimale. Ces valeurs sont considérées par Java comme des valeurs de type `int`. On peut aussi avoir des littéraux de type `long`; il suffit de faire suivre la valeur par la lettre `l` ou `L`.

Pour spécifier un nombre sous forme octale, on utilise `0` comme premier chiffre. Pour spécifier un nombre sous forme hexadécimale, on le commence par `0x` ou `0X` et on peut utiliser indifféremment les lettres `a` à `f` en majuscule ou en minuscule.

Exemple

```
/ Les littéraux suivants sont de type int et représentent l'entier 45
45 // en décimal
055 // en octal
0x2d // en hexadécimal

// Les littéraux suivants sont de type long et représentent l'entier 45
45L // en décimal
055L // en octal
```

0x2dL // en hexadécimal

En ce qui concerne les nombres flottants, ils sont considérés par défaut comme étant des valeurs de type `double`. On peut ajouter la lettre `f` ou `F` derrière le nombre pour qu'ils soient de type `float` et `d` ou `D` pour qu'ils soient de type `double`,

On peut également représenter un nombre flottant en utilisant la notation scientifique. On fait suivre le nombre de `e` ou `E` suivi de la valeur de l'exposant qui peut être positif ou négatif.

Exemple

```
// Les littéraux suivants sont de type double et représentent 25.5
25.5 // en notation décimale
2.55e+1 // en notation scientifique

// Les littéraux suivants sont de type float et représentent 25.5
25.5F // en notation décimale
2.55e+1F // en notation scientifique

// Le littéral suivant est de type double et représente 25.0
25D
```

Caractères

Les caractères sont un autre type fondamental en Java. Ils sont *encodés* d'une certaine manière. Il faut tout d'abord choisir combien de bits on veut utiliser pour représenter un caractère et ce nombre va définir le nombre total de caractères représentables. Ensuite, il faut une table qui fasse le lien entre une séquence de bits et le caractère qu'il représente

un encodage courant dans les pays anglophones est l'ASCII (American Standard Code for Information Interchange). Chaque caractère est encodé sur 7 bits, c'est-à-dire que chaque caractère possède une valeur comprise entre 0 et 127. On peut donc encoder 128 caractères différents avec cet encodage. La figure suivante montre les 128 caractères représentables avec ASCII.

Pour retrouver la valeur d'un caractère, on prend sa ligne et puis sa colonne et on a sa valeur en hexadécimal. Par exemple, le caractère `N` a pour valeur `4E` ou 78 en décimal.

Les caractères sur fond vert sont des *caractères de contrôle* ou caractères non-imprimables, car ils n'ont pas de symbole précis pour les représenter, mais ils peuvent néanmoins être stockés comme toute autre valeur. On a par exemple le saut de ligne (`CR`).

ASCII a été étendu sur 8 bits (iso-8859-1 ou iso-latin-1) ce qui étend le nombre de caractères représentables à 256. Les caractères supplémentaires par rapport à ASCII sont, entre autres, les lettres accentuées non utilisées par les anglophones. Cependant, malgré cette extension, certains caractères de langues asiatiques, arabes ne peuvent être représentés. C'est pourquoi a été créé l'*Unicode*, qui utilise 16 bits et qui permet de représenter 65 536 caractères uniques. C'est cet encodage que les programmeurs de Java ont choisi pour les caractères.

Le type primitif `char` permet de représenter des caractères au format Unicode et sont encodés sur 16 bits.

Littéraux caractères

Un littéral de type caractère est simplement le caractère voulu entouré de guillemets simples (`'`). Il existe des séquences spéciales appelées *séquences d'échappement* qui permettent de représenter des caractères

Exemple

Séquence	Unicode	Caractère
`\n`	000A	nouvelle ligne
`\r`	000D	retour chariot
`\t`	0009	tabulation
`\b`	0008	backspace
`\f`	000C	form feed
`\XXX`		le caractère dont la valeur en octal est XXX
`\uXXXX`		le caractère dont la valeur en hexadécimal est XXXX

Booléens

Le dernier type primitif permet de représenter des *données booléennes*, c'est-à-dire des données qui ne peuvent prendre que deux valeurs différentes : *vrai* ou *faux*. Les littéraux booléens sont : `true` qui représente la valeur vrai et `false` qui représente la valeur faux.

Variables et constantes

Variable

Le concept de *variable* est un concept très important en programmation. Lorsqu'on a besoin de manipuler une valeur, on va travailler avec des variables. Une variable, c'est un genre de boite dans laquelle on peut placer une valeur. Cette boite, on va lui donner *un nom* afin de pouvoir l'utiliser. La

La variable, c'est la boite verte. Le nom de la variable, c'est `variable` et enfin, la valeur de la variable, c'est 17.

Toute variable possède aussi *un type*, c'est le type de la donnée qu'elle stocke.

Déclaration

Java est un langage typé statiquement donc avant de pouvoir utiliser une variable, faut la *déclarer*. Déclarer une variable correspond à dire quel est le type de donnée qu'elle va contenir et lui donner un nom.

Exemple : int machin ;

Cette instruction déclare donc une nouvelle variable, mais elle ne contient rien pour l'instant. On dit que la valeur de la variable est *indéfinie* mais *non-initialisée*.

Modifier le contenu d'une variable

Pour modifier le contenu d'une variable, il faut utiliser *un opérateur*. Par exemple '*opérateur d'affectation* (=) permet de donner une valeur à une variable.

Exemple machin=10 ;

Une variable ne peut contenir que des valeurs qui correspondent à son type. Toute tentative d'affecter à une variable une valeur ne correspondant pas à son type se soldera par une erreur de compilation. De plus, une variable ne peut contenir qu'une seule donnée. Par conséquent, lorsque vous affectez une nouvelle valeur à une variable, l'ancienne valeur est écrasée et supprimée au profit de la nouvelle.

En Java, on peut combiner la déclaration et l'initialisation d'une variable en une seule ligne

Exemple : int machin=10 ;

Afficher le contenu d'une variable

Les méthodes `print` et `println` de l'objet `System.out` pour afficher le contenu d'une variable. Mais si la variable qui n'est pas initialisée, cela provoquera une erreur de compilation.

Constante

Il existe en Java des *constantes* qui sont des variables dont on ne peut changer la valeur qu'elles contiennent. Pour signaler qu'une variable est une constante, on utilise le mot réservé `final` lors de sa déclaration

Exemple : final int machin = 10 ;

La tentative de modifier une constante provoque une erreur à la compilation.

Convention de nommage

Le nom d'une variable est un identificateur, il faut donc qu'il respecte certaines règles de nommage. Par convention, les noms de variables commencent par une minuscule et ne contiennent des majuscules que à chaque changement de mot pour des noms composés (ce qu'on appelle *casse chameau* ou camelcase). Pour les constantes, le nom est complètement en majuscules avec les mots séparés par des tirets de soulignements (_).

Exemples :

```
int vitesseMoteur; // La vitesse du moteur
final int NB_ROUES; // Le nombre de roues
```

Opérateurs

Dans un programme, pour manipuler des données qui sont stockées dans des variables, des constantes et des littéraux pour effectuer des *opérations* sur ces données, comme par exemple des les additions, des multiplications, ...Il faut utiliser des *opérateurs* qui permettent d'effectuer une *opération* bien définie sur des valeurs, appelées *opérandes*, en produisant un résultat appelé *valeur* qui est une donnée d'un certain type.

Il y a de nombreux opérateurs en Java classés de différentes manières comme les opérateurs *unaires*, les opérateurs *unaires*,les opérateurs *binaires* ou les opérateurs *ternaires*.

Opérateur arithmétique

Les *opérateurs arithmétiques* effectuent des opérations arithmétiques sur leurs opérandes telles l'addition ou la soustraction. En plus des quatre opérateurs traditionnels, nous retrouvons un opérateur pour changer le signe d'un opérande et un opérateur qui calcule le reste de la division entière.

Opérateur	Description	Exemple	
binaire	`+`	addition	`x + y`
	`-`	soustraction	`x - y`

	*	multiplication	x * y
	/	division	x / y
	%	modulo	x % y
unaire	-	changement de signe	- x
	+	signe plus	+ x

Lorsqu'on utilise l'opérateur de division (/) avec des opérandes de type entiers, le résultat est donc un entier de type long ou int et il s'agit donc d'une division entière . L'opérateur modulo (%) appliqué à deux entiers calcule le reste de la division entière. Ces deux opérateurs génèrent tous les deux une erreur d'exécution de type `ArithmeticException: / by zero` lorsqu'ils sont appliqués à deux entiers et que l'opérande de droite vaut 0

Opérateur d'égalité et de comparaison

Java propose aussi *des opérateurs d'égalité et de comparaison* qui renvoient un résultat booléen (`true` ou `false`) utilisés pour prendre des décisions dans un programme

les *opérateurs d'égalité* == et != qui sont des opérateurs binaires et testent si les deux opérandes, de type nombres, des caractères ou des booléens, sont égaux ou différents. On peut utiliser ces opérateurs pour comparer des. Cependant les *opérateurs de comparaison* ne sont utilisés qu'avec des nombres.

Opérateur	**Description**	**Exemple**	
égalité	==	égal	x == y
	!=	différent de	x != y
comparaison	<	plus petit que	x < y
	<=	plus petit ou égal que	x <= y
	>	plus grand que	x > y
	>=	plus grand ou égal que	x >= y

Opérateurs logiques

Les *opérateurs logiques* prennent comme des opérandes de type booléens et produisent un résultat de type booléen.

pérateur	**Description**	**Exemple**	**Résultat**	
unaire	!	NON logique	! a	`true` si `a` vaut `false` `false` si `a` vaut `true`
	&&	ET logique	a && b	`true` si `a` et `b` valent `true` `false` sinon
binaire	||	OU logique (inclusif)	a || b	`false` si `a` et `b` valent `false` `true` sinon
	^	OU logique (exclusif)	a ^ b	`false` si `a` et `b` sont égaux `true` sinon

Opérateur d'incrémentation et de décrémentation

L'*opération d'incrémentation* (++) est un opérateur unaire qui ne peut s'appliquer qu'à des variables contenant des nombres (entiers ou flottants) et permet d'ajouter 1 à la variable sur laquelle il est

appliqué. Exemple : x++ ;

Il y a également un *opérateur de décrémentation* (`--`) qui fonctionne comme l'opérateur d'incrémentation qui consiste à retirer 1 de la variable sur laquelle il est appliqué. Exemple : x-- ;

Les deux opérateurs existent sous deux formes. La *forme suffixe* où l'opérateur est placé après la variable, et la *forme préfixe* où l'opérateur est placé avant la variable.

La différence entre les deux formes concerne le moment auquel l'effet de bord est appliqué. Avec la forme préfixe, il est appliqué avant que l'opérateur ait fait son opération. Pour la forme suffixe, l'opération est effectuée et l'incrémentation ou la décrémentation est ensuite appliqué.

Exemple : Pour x=1 ; t=4 ; y= x++ ;z = --t ;

le résultat est x=2 ; y=1 ; t=3 et z=3

Opérateurs d'affectation

L'opérateur d'affectation (`=`) est utilisé pour initialiser une variable ou modifier sa valeur. C'est un opérateur binaire dont l'opérande de gauche doit être une variable et permet de changer sa valeur avec celle de l'opérande de droite qui correspond également au résultat de l'opération et qui détermine le type du résultat.

Exemple s = a+b ;

Opérateur conditionnel

L'*opérateur conditionnel* (`? :`) est le seul opérateur ternaire. Il nécessite donc trois opérandes. Le premier doit être de type booléen et le résultat de l'opération dépend de sa valeur. Si la valeur du premier opérande vaut `true`, le résultat de l'opération correspond à la valeur du second opérande ; sinon, il correspond à celle du troisième. Voyons cela avec l'exemple du listing suivant.

Exemple : min = x<y ? x : y ;// si x<y alors min = x, sinon min = y

Expression

Avec les opérateurs cités nous pouvons construire des *expression* qui sont des combinaisons d'opérateurs avec des opérandes. Une expression va pouvoir être *évaluée*, c'est-à-dire qu'on va pouvoir calculer sa valeur et son type.

Expression simple

Les *expressions simples* n'impliquent aucun opérateur, il y en a trois : les littéraux, les variables et les constantes. La valeur d'un littéral, c'est le littéral, et la valeur d'une variable ou d'une constante, c'est la donnée qui y est stockée. Exemple int x=10, final int y=x ;

Expression complexe

des *expressions complexes* en combinant des opérandes avec des opérateurs et en utilisant des expressions simples. Exemple int x=1, y=2 ; int y= --x+ y++

Affectation multiple

il est possible d'affecter une même valeur à plusieurs variables en une seule opération.

Exemple int x, y, z ; y=z=x=0 ;

Priorité et associativité

Priorité

Les opérateurs Java sont classés selon un ordre de priorité. Toutes les expressions sont évaluées en

suivant la *règle de priorité des opérateurs*. Si on veut forcer une certaine partie d'une expression a être évaluée en premier, il faut utiliser des parenthèses. En effet toute expression parenthésée sera évaluée en premier.

Le tableau ci-dessous liste les opérateurs Java selon leur priorité.

Niveau de priorité	Opérateur	Description	Exemple
1	`++`	incrémentation postfixe	`x++`
	`--`	décrémentation postfixe	`x--`
2	`++`	incrémentation préfixe	`++x`
	`--`	décrémentation préfixe	`--x`
	`-`	changement de signe	`-x`
	`+`	signe	`+x`
	`!`	NON logique	`! x`
4	`*`	multiplication	`x * y`
	`/`	division	`x / y`
	`%`	modulo	`x % y`
5	`+`	addition	`x + y`
	`-`	soustraction	`x - y`
7	`<`	plus petit que	`x < y`
	`<=`	plus petit ou égal que	`x <= y`
	`>`	plus grand que	`x > y`
	`>=`	plus grand ou égal que	`x >= y`
8	`==`	égal	`x == y`
	`!=`	différent de	`x != y`
10	`^`	OU logique exclusif	`x ^ y`
12	`&&`	ET logique	`x && y`
13	`||`	OU logique inclusif	`x || y`
14	`? :`	opérateur conditionnel	`x ? y : z`
15	`=`	affectation	`x = y`
	`+=, -=, *=, /=, %=`	affectations composées	`x += y`

Associativité

Dans le cas où les expressions complexes ne contiennent que des opérateurs qui ont la même priorité., il faut appliquer les règles d'associativité. La plupart des opérateurs sont associatifs de gauche à droite sauf ceux du niveau de priorité 2, 3, 14 et 15 qui le sont de droite à gauche

Types de conversion de donnée primitive

Il y a deux catégories de conversions. Les conversions sans perte d'information et celles qui impliquent une perte d'information. Il n'y a pas de perte d'information lorsque la grandeur du nombre converti est conservée.

Il y a 19 *conversions sans perte d'informations* qui sont reprises dans le tableau ci-dessous. Ces conversions ne perdent jamais d'information du point de vue de la quantité, mais les conversions vers un type primitif décimal peuvent mener à une *perte de précision*. Par exemple, lorsqu'on convertit un `int` ou un `long` vers un `double` ou un `float`, les chiffres les moins significatifs peuvent être perdus alors que la grandeur du nombre est la même.

De	Vers

`byte`	`short`, `int`, `long`, `float` ou `double`
`short`	`int`, `long`, `float` ou `double`
`char`	`int`, `long`, `float` ou `double`
`int`	`long`, `float` ou `double`
`long`	`float` ou `double`
`float`	`double`

Il y a 22 *conversions avec perte d'information*. Les pertes de précision ne se produisent pas dans tous les cas. Elles sont dues au fait qu'on passe d'un type de donnée qui utilise *n* bits vers un autre qui en utilise *m*, avec *m* étant strictement plus petit que *n*.

De	**Vers**
`short`	`byte` ou `char`
`char`	`byte` ou `short`
`int`	`byte`, `short` ou `char`
`long`	`byte`, `short`, `char` ou `int`
`float`	`byte`, `short`, `char`, `int` ou `long`
`double`	`byte`, `short`, `char`, `int`, `long` ou `float`

Structures de contrôles

Instructions conditionnelles

Les *instructions conditionnelles,* ou instructions de test, permettent de faire des choix dans un programme. Elles permettent d'altérer le déroulement du programme en fonction de conditions. Il y a trois instructions conditionnelles différentes en Java : l'instruction `if`, l'instruction `if-else` et l'instruction `switch`.

Instruction `if`

L'*instruction* `if` se retrouve dans de nombreux langages de programmation et permet d'exécuter une instruction si une *condition* est vérifiée. La condition est une expression booléenne et elle est dite vérifiée si sa valeur est `true`

Exemple : `if (x<y) system.out.println(''le min est x= '',x) ;`

Instruction `if-else`

Parfois, on aimerait exécuter une instruction si une condition est vérifiée et une autre instruction si la condition n'est pas vérifiée. Une solution pour ce faire est d'utiliser deux instructions if avec la condition de la seconde qui est la négation de la condition de la première.

Exemple :
```
if (x<y) system.out.println(''le min est x= '',x) ;
         else system.out.println(''le min est y= '',y) ;
```

Instruction `switch`

La dernière instruction conditionnelle disponible en Java est l'*instruction* `switch`. Cette instruction représente un aiguillage multiple : le choix est fait en fonction de la valeur d'une variable ; chaque chemin possible correspond à une valeur différente pour la variable.

Exemple :

```
char x='1 '
switch (x)
{ case '1' : system.out.println(''un'') ; break ;
  case '2' : system.out.println(''deux'') ; break ;
  default : system.out.println(''différent de un et deux '') ;
}
```

La valeur de l'expression du `switch` est tout d'abord évaluée le programme se rend directement au `case` correspondant à cette valeur. L'instruction `System.out.println ("un");` est donc exécutée et puis c'est au tour de l'instruction `break;` celle-ci permet de quitter directement le `switch`. Le cas `default` correspond donc au cas où aucun case ne correspond à la valeur de l'expression

Il s'agit ni plus ni moins d'un raccourci d'écriture pour l'instruction `if-else`.

Instructions répétitives

Les *instructions répétitives*, également appelées *instructions itératives* ou *boucles*, permettent de répéter un certain nombre de fois une instruction ou un bloc de code. Il y a quatre types de boucle en Java : l'instruction `while`, l'instruction `do...While`, l'instruction `for` et l'instruction *for-each* nous abordons que les trois premières, et nous parlerons de la quatrième dans les collections.

Instruction `while`

L'*instruction* `while` permet de répéter une instruction ou un bloc de code tant qu'une condition est vraie. La condition est une expression booléenne, comme pour l'instruction `if-else`.

L'exécution d'une boucle est une succession d'*itérations*. Une itération correspond donc à une exécution du corps de la boucle.

`Exemple` :
```
Int x=1 ;
While (x<10)
     system.out.println(''x= '',x++) ;
```

Instruction do

L'*instruction* `do while` est similaire à l'instruction `while` sauf pour ce qui est de l'ordre dans lequel les choses sont exécutées. Le corps de la boucle est d'abord exécuté puis la condition est évaluée. Si sa valeur est `false`, le programme continue; sinon, le corps de la boucle est à nouveau exécuté, etc ... Ceci implique donc que le corps de la boucle sera exécuté au moins une fois dans tous les cas. Faites bien attention qu'il faut un point-virgule (`;`) après la condition.

Exemple :
```
Int x=1 ;
do{
     system.out.println(''x= '',x++) ;
}While (x<10)
```

Instruction `for`

Les deux instructions que l'on vient de voir sont intéressantes si vous ne savez pas de manière explicite combien de fois vous voulez que le corps de la boucle soit exécuté. Si l'on veut exécuter une instruction ou un bloc de code un nombre précis et connu de fois, on va préférer utiliser l'*instruction* `for`.

l'instruction `for` commence par le mot réservé `for` suivi de trois éléments séparés par des points-virgules et mis entre parenthèses et vient ensuite le corps de la boucle. Les trois éléments de la boucle `for` sont les parties *initialisation*, *condition* et enfin *mise à jour* et elles sont toutes optionnelles.

Exemple :

```
for(x=1;x<10 ;x++)
      system.out.println(''x= '',x++) ;
```

On peut déclarer et initialiser plusieurs variables du même type dans la partie initialisation et 'on peut avoir plusieurs instructions dans la partie mise à jour. Il suffit de les séparer par une virgule.

Instructions de branchement

Les *instructions de branchement* permettent d'interrompre le déroulement normal du programme en faisant un saut d'un endroit du programme vers une autre endroit. C'est pourquoi on les appelle également *instructions de saut*. Il y a trois instructions de branchement en Java : les instructions `break`, `continue` et `return`.

Instruction `break`

l'*instruction* `break` vu avec l'instruction `switch` peut également être utilisée dans le corps de n'importe quelle boucle. L'instruction a pour effet de quitter directement le `switch` ou la boucle et de poursuivre le programme à l'instruction suivante.

Instruction `continue`

L'instruction `continue`, utilisable uniquement dans les boucles, permet d'arrêter l'exécution de l'itération en cours et de passer directement à l'itération suivante. Dans le cas des boucles `while` et `do while`, le progreamme revient directement à la vérification de la condition et pour les boucles `for`, e programme passe à la partie mise à jour.

Instruction `return`

L'*instruction* `return` s'utilise avec les méthodes pour retourner ou renvoyer une valeur du même type.

Étiquettes

Lorsqu'on a plusieurs boucles imbriquées les unes dans les autres, et que l'on utilise une instruction `break` ou `continue` dans la boucle interne (la boucle qui est à l'intérieur du corps de l'autre), seule cette dernière sera altérée, mais la boucle externe continuera. Grâce aux *étiquettes*, on peut préciser quelle boucle doit être affectée

Exemple

```
etiq1: while (pasFini) { // etiq1: étiquette
                ...
                for (int i=0; i < n; i++) {
                      ...
                      if (t[i] < 0)
                            continue etiq1; // ré-exécuter à partir de etiq1
                            ...
                }
                ...
     }
```

Le concept Objet de Java

Classes

Une classe est un type structuré de données (prolongement des structures C : *struct*). Elle de modéliser et décrire un ensemble des objets qui ont des propriétés et des comportements communs. Un programme Java utilise certaines classes prédéfinies dans la bibliothèque Java. Ces classes sont rassemblées par catégorie (packages), ces dérniers doivent être spécifiés avant leur utilisation par l'instruction `import` sauf pour le package `java.lang` qui est utilisé par défaut et ilcontient les définitions des classes les plus usuelles.

Déclaration d'une classe Java

Pour déclarer une classe (un nouveau type d'objets) on utilise le mot clé class suivi du nom de la classe.
La syntaxe pour créer une classe de nom "NomClasse" est la suivante:

```
class NomClasse{ // NomClasse est le nom de la classe à créer */
                 /* Le corps de la classe comprend :
                 - la définition des attributs (données membres)
                 - la définition des méthodes et des constructeurs. */
}
```

Pour les commentaire on utilise:

// pour un commentaire sur une seule ligne
/* pour un commentaire étalé sur
plusieurs lignes. Il doit terminer avec */

Attributs

La classe contient des attributs appelés aussi champs ou données membres, définissent l'état interne d'un objet. Ilsqui peuvent être soient des variables (ou des constantes) de type primitifs (booléen, caractère, entier, réel). Ces variable ne sont pas obligatoirement initialisées auquel cas elle prennent la valeur *false* pour les booléen et 0 pour les autres types. Soient des références vers d'autres objets (une référence n'est pas un objet mais elle permet de manipuler l'objet référencé). L'objet référencé est mémorisé dans une zone de mémoire spéciale appelée TAS gérée par la machine Java. De plus, la référence n'est pas initialisée (sauf initialisation explicite) et prend la valeur `null` (pour un attribut) et ne référence aucun objet. Ces attributs correspondent aux propriétés de l'objet et doivent être déclarés avant d'être utilisés..

Syntaxe:

modificateur type nomAttribut

- modificateur : un spécificateur d'accès.
- nomAttribut : le nom de l'attribut (le nom de la variable).
- type : le type de l'attribut. Il est de type primitif (char, byte, short, int, long, float, double, boolean) ou un nom d'une classe.

Exemple :

```
private double x ;  // nom de l'attribut : x, le type : double ; le modificateur : private
public Point A ; // nom de l'attribut : A, le type : Point ; le modificateur : public
```

Méthodes

La classe contient aussi des méthodes appelées aussi fonctions membres ou comportement définissent l'ensemble d'opérations applicables à l'objet (les objets sont manipulés avec les appels de ces méthodes). Une méthode est soit :

 Une fonction ou une procédure : Une fonction est un sous-programme retournant une valeur, une procédure n'en retourne pas. Fonctions et procédures peuvent avoir des paramètres composés de données de type primitif et/ou de références d'objets. Lors de la définition, ces paramètres sont appelés paramètres formels.

 Soit un **constructeur** avec ou sans paramètres : Un constructeur est une méthode spéciale portant le nom de la classe et ayant pour but d'initialiser (on dit construire) l'objet. Le constructeur est appelé lors de l'instanciation de l'objet. Le constructeur utilisé est déterminé par le nombre et les types de ces paramètres précisé lors de l'instanciation de l'objet.
Si une classe ne définit aucun constructeur, un **constructeur par défaut** (constructeur sans paramètre) est automatiquement défini. Dans le cas contraire, le constructeur par défaut n'est pas défini (sauf définition explicite).

Syntaxe:

```
modificateur typeRetour nomMethode ( listeParamètres ) {
        // corps de la méthode: les instructions décrivant la méthode
}
```

- modificateur : un spécificateur d'accès.
- nomMethode : le nom de la méthode
- listeParamètres : la liste (éventuellement vide) des paramètres (arguments). On spécifie les types et les noms des informations qu'on souhaite passer à la méthode lors de son appel.
- typeRetour : c'est le type de la valeur qui sera retournée par la méthode après son appel. Si la méthode ne fournit aucun résultat, alors typeRetour est remplacé par le mot clé void. Dans le cas où la méthode retourne une valeur, la valeur retournée par la fonction doit être spécifiée dans le corps de la méthode par l'instruction de retour:

```
return expression;
```

ou

`return;` // possible uniquement pour une fonction de type void

Signature d'une méthode en java

La signature d'une méthode permet d'identifier de manière unique une méthode au sein d'une classe. En java, la signature d'une méthode est l'ensemble composé de :
• du nom de la méthode
• de l'ensemble des types de ses paramètres.
Mais attention, en java, le type de retour ne fait pas partie de la signature de la méthode.

Passage des paramètres

Le mode de passage des paramètres dans les méthodes dépend de la nature des paramètres :

- **par valeur** pour les types primitifs.
- **par valeur des références** pour les objets: la référence est passée par valeur (i.e. le paramètre est une copie de la référence), mais le contenu de l'objet référencé peut être modifié par la méthode (car la copie de référence pointe vers le même objet).

Surcharge des méthodes

On parle de surcharge (en anglais overloading) lorsqu'un même symbole possède plusieurs significations. Le choix entre ces symboles se fait en fonction du contexte. Par exemple le symbole +

dans l'instruction « a+b » dépend du type des variables a et b. En Java la surcharge des opérateurs comme en C++ est impossible. Cependant la surcharge des méthodes est possuble, c'est-à-dire on peut définir, dans la même classe, plusieurs méthodes qui ont le même nom mais pas la même signature. Elles sont différenciées par le nombre et/ou les types des arguments et aussi les types de retour .

Exemple : Considérons l'exemple suivant ou la classe "`Classe1` " contient 3 méthodes du même nom "Somme"

- La première a deux arguments de type `int`. Elle calcule la somme des deux valeurs.

- La deuxième a trois arguments de type `int`. Elle calcule la somme des trois valeurs.
- La troisième a deux arguments de type `double`. Elle calcule la somme des deux valeurs.

```
class Classe1{
      public int somme(int x, int y) {
            return (x+y);
      }
      public int somme(int x, int y,int z) {
            return (x+y+z);
      }
public double somme(double x, double y) {
            return (x+y);
      }
}
public class TestClasse1 {
      public static void main(String[] args) {
            Classe1 obj1;
            System.out.println("Somme1="+obj1.somme(9, 3));
            System.out.println("Somme2="+obj1.somme(9, 3,5));
            System.out.println("Somme3="+obj1.somme(9.0, 3.1));
      }
}
```

Conventions de nommage

 Une classe commence toujours par une majuscule. Si la classe est un mot composé, la première lettre du mot composé est en majuscules. Exemple `Moteur, MoteurAvion`

 Un attribut commence toujours par une minuscule. Si l'attribut est un mot composé, la première lettre du mot composé est en majuscules. Exemple : `tableau`, `unTableau`. Une constante est toujours en majuscules.

 Une méthode commence toujours par une minuscule. Si la méthode est un mot composé, la première lettre du mot composé est en majuscules.

Création et manipulation d'Objets

Une fois la classe est définie, on peut l'utiliser pour créer des objets (variables). Donc chaque objet est une variable ou instance d'une classe qui a son espace mémoire pour stocker les valeurs de ses attributs qui sont propres à l'objet (appelés état interne de l'objet).
L'opération de création de l'objet est appelée une instanciation, l'objet est donc référencé par une variable ayant un état (une valeur). Une classe permet d'instancier plusieurs objets. De telle façon que les noms des attributs et les méthodes sont les mêmes pour toutes les instances de la classe ; mais les valeurs des attributs sont propres à chaque objet.
Contrairement aux types primitifs (dont la déclaration réserve un emplacement mémoire pour stocker la valeur de la variable), la création d'objets se passe en deux étapes, déclaration avec son type qui permet de référencer l'objet, ensuite allocation de l'emplacement mémoire (les objets sont manipulés avec des références)

Syntaxe de déclaration:

NomClasse idObjet1, idObjet2, … ;

- NomDeClasse : désigne le nom de la class.
- idObjet1, idObjet2, … : désignent les noms des variables de type "NomClasse " qui sont des variables pour référencer un objet de type cette classe**.** Aucun objet n'est créé, ils sont seulement déclarés, ils vallent "null" et ne peuvent être utilisés.

Après la déclaration d'un objet, il faut réserver un emplacement mémoire pour stocker l'état interne de l'objet correspondant. La création doit être demandée explicitement dans le programme en faisant appel à l'opérateur « `new` »
qui permet de réserver de la place mémoire pour stocker l'objet et initialiser les attributs.

Syntaxe :

nomClasse idObjet;
idObjet = new nomClasse();

Les deux expressions peuvent être remplacées par :

NomClasse idObjet = new NomClasse();

Exemple1 :

Considérons la classe rectangle

```
public class Rectangle{
      int longueur;
      int largeur;
      int x;
      int y;
      public static void main (String args[]) {
            Rectangle rectangleR1; /* déclare une référence sur l'objet rectangleR1 */
            rectangleR1 = new Rectangle(); /* création de l'objet rectangleR1 */
            // rectangle R1 est une instance de la classe Rectangle
            // Les deux expressions peuvent être remplacées par :
            // Rectangle rectangleR1=new Rectangle();
      }
}
```

Initialisation par défaut

La création d'un objet entraîne toujours une initialisation par défaut de tous les attributs de l'objet. Cette garanti d'initialisation par défaut ne s'applique pas aux variables locales (variables déclarée dans un bloc: par exemple les variables locales d'une méthode).

Accès aux attributs

Une fois l'objet est créé, on peut accéder à ses attributs (données membres) en indiquant le nom de l'objet suivi par un point, suivi par le nom de l'attribut comme suit:

nomObjet.nomAttibut

nomObjet : nom de la référence à l'objet (nom de l'objet)
nomAttribut : nom de la donnée membre (nom de l'attribut).

Accès aux méthodes: appels des méthodes

- Une méthode ne peut être appelée que pour un objet.
- L'appel d'une méthode pour un objet se réalise en indiquant le nom de l'objet suivi d'un point, suivi du nom de la méthode et de sa liste d'arguments:

nomObjet.nomMethode(listeArguments).

- nomObjet: nom de l'objet
- nomMethode: nom de la méthode.
- listeArguments : liste des arguments

Exemple:

Considérons la classe **Class1** suivante:

```
class Classe1 {
      double x;
      boolean b;
      void init(double x1, boolean b){
            x=x1 ;
            b=b1 ;
      }
}

class Classe1App {
      public static void main (String args[]) {
            Classe1 obj = new Classe1(); /* crée un objet de référence obj
                                                (crée un objet : obj) */
            obj.b=true; // affecte true à l'attribut b de l'objet obj.
            obj.x=2.3; // affecte le réel 2.3 à l'attribut x de l'objet obj.
            obj. init(0,false); // affecte 0 à l'attribut x et false à l'attribut y de l'objet obj
      }
}
```

Concept d'encapsulation

Définition

Un des principes de la programmation orientée objet consiste à masquer au maximum les détails d'implémentation, c'est-à-dire rendre l'utilisation d'une classe indépendante de son implémentation (codage). En pratique on doit pouvoir modifier une classe sans affecter les programmes utilisant cette classe. En Java, l'encapsulation est assurée par les modificateurs d'accès (spécificateurs d'accès) permettant
de préciser 3 niveaux de visibilité des membres de la classe. La portée est spécifiée en utilisant les préfixes <rien>, `private`, `protected`, `public` qui définissent les droits d'accès aux données suivant le type d'accès:
- Membres publiques (membres précédés par le modificateur "**public** "): correspond à la partie visible de l'objet depuis l'extérieur et donc utilisable par une classe éxterieure
- Membres privés (membres précédés par le modificateur "**private**"): correspond à la partie non visible de l'objet qui n'est accessible que depuis l'intérieur de la classe elle même, c'est à dire seulement les méthodes de la classe ont le droit d'accéder directement aux membres privés.
- Membre par défaut: lorsqu'aucun spécificateur d'accès n'est mentionné devant un membre d'une classe, on dit que ce membre a l'accessibilité par défaut. Ce membre n'est accessible que depuis sa propre classe et depuis les autres classes du même package.
- Membres protégés (membres précédés par le modificateur "**potected**"): un membre protégé se comporte comme un membre privé avec moins de restriction: il a une portée locale à la classe, au package et aux classes qui héritent de la classe du membre **(**une sous-classe a un accès direct aux membres **protected** mais pas aux membres **private)**.

Règles d'encapsulation

 Les attributs doivent êtres dans la majorité des cas privés (`private`). Pour accéder à un attribut depuis l'extérieur de la classe créer un assesseur, c'est-à-dire une fonction permettant de lire et/ou de modifier l'attribut .

 Les méthodes accessibles depuis l'extérieur de la classe sont soit publiques, si l'accès doit se faire en-dehors de package, soit sans spécification de portée lorsque l'accès doit être locale au package (cas très particulier).

 Les méthodes exclusivement utilisées par les fonctions membre de la classe doivent avoir une portée `private`.

 Un constructeur doit avoir une portée `public`.

 Si une méthode ou un attribut doit être hérité, la portée doit être spécifiée avec l'attribut `protected`.

 Si la classe est préfixée avec l'attribut `public` sa portée est globale, sinon elle est locale au package dans lequelle elle a été définie.

Exemple 1: Accès pour modification d'un attribut depuis l'extérieur.

```
class Classe1 {
      public int x;
      private int y;
      public void initialise (int i, int j){
            x=i;
            y=j;  /* Valide: accès de puis l'intérieur à l'attribut private y*/
      }
}
public class Test1{  // fichier source de nom Test1.java
      public static void main (String args[]) {
            Classe1 obj1 = new Classe1();
            obj1.initialise(1,3);  /* valide : accès direct depuis une autre classe à une
```

```
                                                    méthode public */
            obj1.x=2;  /* Valide: accès direct depuis l'extérieur à un attribut public x. la valeur
                          de x de l'objet obj1 devienne 2 */
            objA.y=3;  /* Non valide : accès direct depuis une classe externe à un attribut
                          déclaré private */
}
}
```

Exemple2: Accès pour consultation d'un attribut depuis l'extérieur: affichage du contenu d'un attribut.

```
class Classe2 {
      public int x;
      private int y;
      public void initialise (int i, int j){
            x=i;
            y=j;  // Valide: accès de puis l'intérieur à l'attribut private y
      }
}
public class Test2{  // fichier source de nom Test2.java
      public static void main (String args[]) {
            Classe2 obj2 = new Classe2();
            obj2.initialise(1,3);  // Valide : car la méthode initialise() est déclarée public
            System.out.println(" x= "+obj2.x);  // Valide : l'attribut x déclaré public.
            System.out.println(" y= "+obj2.y);  /* Non valide : accès depuis
                                                   l'extérieur l'attribut private y */
      }
}
```

Méthodes d'accès aux variables depuis une classes externe

Pour accéder en lecture (par exemple afficher) la valeur d'un attribut protégé (private), on définit ***Un accesseur*** (***un getter***): c'est une méthode qui permet d'accéder en lecture au contenu de l'attribut, c'est-à-dire, lire depuis une classe externe, le contenu d'une donnée d'un attribut privé. Généralement on lui donne comme nom : **getNomAttribut() ;**

Pour accéder en écriture (modifier) la valeur d'un attribut protégé (private), on définie

Un modificateur (mutateur ou setter): c'est une méthode qui permet d'accéder en écriture à l'attribut,c'est-à-dire modifier, depuis une classe externe, le contenu d'un attribut privé. Généralement on lui donne comme nom : **setNomAttribut() ;**

Exemple:

```
class Etudiant {
      private int cne;
      public int getCNE (){ //un getter
            return(cne);
      }
      public void setCNE (int cne1){ //un setter
            cne=cne1;
      }
}
```

Autoréférences: emploi de this

Au sein d'une méthode, on peut utiliser le mot clé "**this**", pour désigner l'instance courante (l'objet courant) c'est pour faire référence à l'objet qui a appelé cette méthode.

```
public void f(….) {
```

// l'emploi de this , dans ce bloc, désigne la référence à l'objet ayant appelé la méthode f()
}

Exemple:

```
class Etudiant {
      private String nom, prenom;
      public initialise(String n1, String p1) {
            nom = n1;
            prenom = p2;
      }
}
```

Comme les identificateurs `n1` et `p1` sont des arguments muets pour la méthode `initialise`(), alors on peut les noter `nom` et `prenom` qui n'ont aucune relation avec les attributs private `nom` et `prenom`.

Dans ce cas la classe `Etudiant` peut s'écrire comme suit:

```
class Etudiant {
      private String nom, prenom;
      public initialise(String nom, String prenom){
            this.nom=nom;
            this.prenom=prenom;
      }
}
```

`this.nom` désigne l'attribut `nom` de l'objet qui appelle la méthode `initialise`()
`this.prenom` désigne l'attribut `prenom` de l'objet qui appelle la méthode `initialise`()

Attributs et méthodes d'instance ou de classe

Par défaut les attributs et les méthodes sont liés à un objet et sont appelées attributs et méthodes d'instances. Ceci a pour conséquence que chaque objet issu de la même classe possède des attributs indépendants.

Il est possible de casser l'affinité que possède un attribut ou une méthode avec un objet à l'aide du mot ***`static`***. Cet attribut ou cette méthode est alors lié à la classe et non à l'objet dont il est issu et **devient commun** à tous les objets. Ils sont comparables aux "variables globales" et n'existent qu'en un seul exemplaire

Les attributs et méthodes deviennent alors des attributs et des méthodes de classe partagés par toutes les instances d'une classe. ils sont donc commun à tous les objets et peuvent être utilisés pour partager un ensemble de valeurs communes.

L'utilisation du préfixe statique n'est pas anodin et a des conséquences importantes :

- Les fonctions de classes (fonctions statiques) ne peuvent plus accéder aux attributs et aux méthodes d'instances.
- La durée de vie des attributs devient celle du programme et non celle d'un objet particulier.
- Méthode ou attribut peuvent être accédés sans utiliser d'objet en utilisant l'opérateur *point*.

Exemple: Considérons la classe:

```
class Classe1 {
      static int n;  // la valeur de n est partagée par toutes les instances
      double y;
      public static void f() {  // méthode de clase
            /* ici (dans le corps de la méthode f(), on ne pas accéder au champ x, champs usuel
                  Par contre on peut accéder au champ statique n*/
      }
}
Classe1 objA1=new Classe1(),
Classe1 objA2=new Classe1();
```

Ces déclarations créées deux objets (instances) différents: objA1 et objA2.
Puisque la variable n est un attribut précédé du modificateur **static**, alors il est partagé par tous les objets en particulier, les objets objA1 et objA2 partagent la même variable n dont la valeur est indépendante de l'instance (l'objet). Pour accéder au champ statique n, on peut utiliser soit le nom de la classe soit le nom d'une de ses instances (Classe1.n ou objA1.n)

Exemple:
```
public class MethodeStatic{
      public static void main(String[] argv) {
            ClasseTest objA1=new ClasseTest();
            objA1.n+=4;  // objA1.n vaut 4;
                              // équivalent à ClasseTest.n=4;
            ClasseTest objA2=new ClasseTest();
            // objA2.n vaut 4 car « n » est champs de classe (champ statique).
      }
}
```

Les constructeurs

Définition

Le constructeur est une méthode particulière qui permet de créer (instancier) les objets. Il porte le même nom que la classe et n'a pas de type retour. Il peut contenir des instructions à exécuter pendant la création des objets, en particulier, il contient des instructions pour l'initialisation des attributs. Par conséquent la création de l'objet et son initialisation peuvent être unifiées. Quand une classe possède un constructeur, Java l'appel automatiquement à toute création d'objet avant qu'il ne puisse être utilisé.
Chaque classe peut avoir un ou plusieurs constructeurs qui servent à créer les instances et initialiser leurs états. Pour créer une instance d'une classe, on utilise l'opérateur **new** avec l'un des constructeurs de la classe. Cette invocation :
1. alloue l'espace mémoire nécessaire pour stocker les données (les propriétés) de l'objet.
2. crée une référence sur cet espace mémoire.
3. exécute le code du constructeur.

Exemple: Considérons la classe point et transformons la méthode initialise en un constructeur.
```
public class Point{
      private int x;  // abscisse: variable d'instance
      private int y;  // ordonnée: variable d'instance
      public Point (int x, int y) {  // Constructeur qui remplace la méthode initialise()
            this.x=x;  /* x désigne la valeur passé en argument alors que this.x désigne l'attribut
                              x */
            this.y=y;  /* y désigne la valeur passé en argument alors que this.y désigne l'attribut
                        y */
      }
      public void affiche (){
            System.out.println(" abscisse : "+ x + " ordonnée : " + y);
      }
}
public class PointApp{
      public static void main (String args[]) {
            Point pA=new Point(1,1);  // création et initialisation de l'objet pA
            pA.affiche() ;
            Point pB=new Point(2,3);  // création et initialisation de l'objet pB
            pB.affiche();
      }
```

```
}
```
L'instruction Point pA=new Point(1,1); appelle le constructeur Point() définie dans la classe Point pour créer l'objet pA. L'exécution du code du constructeur Point() permet d'initialiser les attributs de l'objet pA.

Constructeur par défaut :

Si la classe ne déclare pas explicitement un constructeur, alors un constructeur par défaut sera automatiquement ajouté par le compilateur Java. Ce constructeurs sert à :
- allouer l'espace mémoire nécessaire pour stocker les données de l'objet
- crée une référence sur cet espace mémoire
- initialise les attributs par défaut (voir initialisation par défaut):

Cependant Le constructeur par défaut ne peut être utilisé pour l'instanciation des objets que lorsque la classe ne possède pas explicitement un constructeur.

Constructeur par recopie

Le constructeur d'une classe peut posséder un unique paramètre qui est une référence à un objet de la classe appelé *constructeur par recopie*. Ce cas est très utilisé car bien souvent, dans un programme, on veut dupliquer un objet (afin, par exemple, de travailler sur sa copie et de garder une sauvegarde). Le constructeur par recopie peut être déclaré de la manière suivante :

public NomClasse(NomClasse) ;

Pour dupliquer l'objet a dans un objet b, le programmeur écrira simplement :

NomClasse objest1 = new NomClasse(objet2) ;

Exemple

```
public Point (Point);
Point b = new Point(a);
```

Règles des constructeurs

Un constructeur :

1. porte le même nom que la classe.
2. ne doit figurer devant son nom aucun type de retour, même void.
3. peut disposer d'un nombre quelconque d'arguments (éventuellement aucun).
4. il doit être appelé à chaque création, lorsque la classe au moins comporte un constructeur.
5. il ne peut pas être appelé de la même manière que les autres méthodes. Par exemple :
`a.Point (2,3)` n'est pas valide car `Point()` est un constructeur et non pas une méthode classique.
6. Lorsque la classe ne comporte pas de constructeur, un constructeur par défaut sera automatiquement ajouté par Java pour créer les objets.
7. Si un constructeur est déclaré private, dans ce cas il ne pourra pas être utilisé, à l'extérieur de la classe, pour instancier les objets.

Initialisation des attributs

On distingue 3 types d'initialisation des attributs:

1. Initialisation par défaut: les champs d'un objet sont toujours initialisés par défaut.
`Boolean (False)` ; `char` (caractère de code nul) ; `int, byte, short, int long (0)` ; `float, double (0.f ou 0.)` et `class (Null)`

2. Initialisation explicite des champs d'un objet

Un attribut peut être initialisé explicitement pendant sa déclaration. Par exemple:

```
class Classe1{
      private int n=10;
```

```
    private int p;
}
```

3. Initialisation par le constructeur

On peut aussi initialiser les attributs lors de l'exécution des instructions du corps du constructeur. Exemple du constructeur `Point(int,int):  Point pA=new Point(8,12);` . D'une manière générale, la
création d'un objet entraîne toujours, par ordre chronologique, les opérations suivantes:

1. l'initialisation par défaut de tous les champs de l'objet.
2. l'initialisation explicite lors de la déclaration du champ.
3. l'exécution des instructions du corps du constructeur.

Surcharge des constructeurs (Plusieurs constructeurs)

Une classe peut avoir plusieurs constructeurs. Comme le nom du constructeur est identique au nom de la classe, il s'agit de la surcharge donc même nom (celui de la classe) mais types et/ou nombre de paramètres différents.

Utilisation du mot clé this dans un constructeurs

Il est possible d'appeler, au sein d'un constructeur, un autre constructeur de la même classe en utilisant l'instruction **`this (arguments) ;`** où arguments désigne les arguments du constructeur appelé. Cette instruction doit être la première instruction.

Concepts avancés de Java

Tableaux

Déclaration et création des tableaux

Les tableaux sont considérés comme des objets. Ils sont manipulés par des références : les variables de type tableau contiennent des références aux tableaux. Ils sont créés par l'opérateur **new:** allocation dynamique par **new.** Les éléments du tableau peuvent être d'un type primitif ou d'un type objet.
La création des tableaux s'effectue en deux étapes :
1. Déclaration : cette étape détermine seulement le type des éléments du tableau.
2. Création et dimensionnement : on détermine la taille du tableau (c'est-à-dire le nombre d'éléments) pendant la création (instanciation) et non pendant lé déclaration.

1. Déclaration :

La déclaration d'une référence à un tableau précise le type des éléments du tableau. Elle s'effectue par l'une des syntaxes suivantes :

type[] id ; ou type id[] ;

Cette dernière forme permet de mélanger des tableaux et des variables de même de type.

- type : précise le type des éléments du tableau.
- id : est une référence à un tableau

Exemple:

```
int [] a, tab1, b; // a, tabNotes et b sont tous des références à des tableaux de type int.
double a, tab1[ ], b; // mélange de tableaux et de variables de même de type.
```

En java la taille des tableaux n'est pas fixée à la déclaration. Elle est fixée quand l'objet tableau sera effectivement créé snon cela provoque une erreur à la compilation

2. Création

La création des tableaux s'effectue avec l'opérateur **new** en précisant sa taille.

```
int [] tab; // déclaration que « tab » est une référence à un tableau d'entiers
tab = new int[5]; /* création effective de l'objet tableau et fixation de sa taille Dans cet
                     exemple on a créé un tableau de 5 éléments de type int. */
```

3. Déclaration et création.

```
int [] tab = new int[5];
```

La création d'un tableau par l'opérateur new:

• Alloue la mémoire en fonction du type et de la taille

• Initialise, par défaut, chaque élément du tableau à 0 ou à false ou à null, suivant le type de base du tableau.

Accès à la taille et aux éléments d'un tableau:

Accès à la taille:

La taille est accessible par le "champ final" length: **final int length**

Exemple :

```
tab = new int[8];
int l = tab.length; // la longueur du tableau tab est l = 8
```

La taille du tableau est définie à l'exécution. La taille peut être une variable. Elle est fixée une fois pour toute. Après exécution, **la taille ne pourra plus être modifiée**.

```
int n=Clavier.lireInt( ); // saisir au clavier l'entier n.
int tab[ ]=new int [n];
```

Par contre la référence `tab` peut changer. Elle peut par exemple référencer un autre objet de taille différente : par affectation ou par `new`.

Exemple:

```
double [] tab1, tab2; // tab1 et tab2 deux tableaux de type double
tab1=new double [10]; // tab1 de taille 10: tab1.length = 10
tab2=new double [15]; // tab2 de taille 15: tab2.length = 15
tab1=new double [36]; /* tab1 référence  maintenant un nouveau tableau de taille 36.
                                          Maintenant tab1.length = 36. */
tab1=tab2 ; /* tab1 et tab2 désignent le même tableau référencé par tab2.
                  Donc tab1.length=15 */
```

Indices du tableau:

Les indices d'un tableau ***tab*** *sont comprises entre* ***0*** et (***tab.length – 1)***. Java vérifie automatiquement l'indice lors de l'accès. Il lève une exception, de type ArrayIndexOutOfBoundsException, s'il y a tentative d'accès hors bornes.

Accès aux éléments du tableau:

Soit tab un tableau. On accède à l'élément d'indice i, pour `0≤i≤(tab.length – 1)`, du tableau tab en écrivant `tab[i]`.

Exemple :

```
int tab[] = new int[8]; // crée un tableau, nommé tab, de type int et de taille 8
int e1=tab[4] ; // accès au 5éme élément du tableau tab (élément d'indice 4)
int e2 = tab[8]; /* tentative d'accès hors bornes, en effet les indices du tableau sont compris entre 0
et 7. L'exécution produit le message ArrayIndexOutOfBoundsException */
```

Autres méthodes de création et d'initialisation.

On peut lier la déclaration, la création et l'initialisation explicite d'un tableau. La longueur du tableau ainsi créé est alors calculée automatiquement d'après le nombre de valeurs données.
• Création et initialisation explicite à la déclaration : (cette syntaxe n'est autorisée que dans la déclaration)

```
int tab[] = {5, 2*7, 8}; /*crée un tableau de taille 3. Les éléments du tableau sont initialisés à 5, 14, 8 */
```

• Initialisation pendant la création.

```
int[] tab;
tab = new int[] {6, 9, 3, 10}; /* crée un tableau de taille 4. Les éléments du tableau sont initialisés à 6, 9, 3, 10. */
```

Exemple de tableau de paramètres de la ligne de commande

```
public class HelloWorld {
     public static void main(String[] args) {
          for (int i=0; i < args.length; i++)
               System.out.println("argument"+i+ " = "+args[i]);
          }
     }
}
```

$ javac `HelloWorld`// compilation
$ java `HelloWorld` Bonjour tout le monde
Exécution du programme `HelloWorld` en passant 4 arguments (quatre chaînes de caractères (mots) à savoir :
Bonjour : premier argument (élément d'indice 0 du tableau args)
tout : deuxième argument (élément d'indice 1 du tableau args)
le : troisième argument (élément d'indice 2 du tableau args)
monde: quatrième argument (élément d'indice 3 du tableau args) */
Le programme affichera:
argument 0 = Bonjour
argument 1 = tout
argument 2 = le
argument 3 = monde

Affectation des tableaux

Java permet de manipuler globalement les tableaux par affectation de leurs références. L'affectation ne modifie que la référence. Soient tab1 et tab2 deux tableaux, l'instruction tab1=tab2;
Ne copie pas les valeurs du tableau tab2 dans tab1, mais plutôt copie la référence contenue dans tab2 est affectée à tab1. Maintenant tab1 et tab2 désignent le même objet tableau qui était initialement référencé par tab2 (l'objet initialement référencé par tab1 est perdu).

Copie des tableaux

Pour copier les valeurs de tab2 dans tab1, il faut soit utiliser une méthode manuelle (copier élément par élément), soit utilise les méthodes des classes prédéfinies:

System.arraycopy(src, srcPos, dest, destPos, nb);

src : tableau source.
srcPos : indice du 1er élément copié à partir de src.
dest : tableau destination
destPos : indice de dst où sera copié le 1er élément
nb : nombre d'éléments copiés

Comparaison de tableaux

Comparer le contenu de deux tableaux de types primitifs:
- On peut comparer les éléments des deux tableaux un à un .
- On peut aussi utiliser la méthode **equals()** de la classe **Arrays :**

`boolean b=Arrays.equals(tab1,tab2);` /* compare les contenus des tableaux tab1 et tab2
et retourne le résultat de la comparaison dans la variable booléenne b */

Tableaux d'objets

Les éléments d'un tableau peuvent être d'un type objet. La déclaration d'un tableau d'objets crée uniquement des «cases» pour stocker des ***références*** aux objets, mais ne crée pas *les objets eux-mêmes.*
Ces références valent initialement **null**. Il faut créer les objets avant de les utiliser.
Dans ce cas la méthode **equals()** de la classe **Arrays** compare les références.

Les tableaux en argument d'une méthode:

Le passage des tableaux se fait par spécification du tableau comme paramètre d'une méthode.
La taille de chaque dimension n'est pas spécifiée, mais peut être obtenue dans la méthode à l'aide de l'attribut *length*. Le passage des tableaux comme paramètres des méthodes se fait par ***référence*** (comme les objets) et non par ***copie*** (comme les types primitifs). La méthode agit donc directement sur le tableau et non sur sa copie.

Tableaux à deux dimensions

Un tableau à deux dimensions, ou matrice, représente un ensemble de lignes et de colonnes. Si tab est un tableau à deux dimensions, l'élément de la ligne i et de la colonne j est désigné par : `tab[i][j]`.

Déclaration
int[][] notes; // déclare un tableau à deux dimensions
Chaque élément du tableau contient une référence vers un tableau

Création:
On utilise l'opération new de création de tableaux. Il faut donner au moins la première dimension

Exemple:
`notes = new int[30][3];` // créé un tableau de 30 étudiants, chacun a au plus 3 notes
Par convention, la première dimension est celle des lignes, et la deuxième, celle des colonnes. Les indices débutent à 0. Lors de sa création, les éléments du tableau sont initialisés par défaut.
`notes = new int[30][];` /*crée un tableau de 30 étudiants, chacun a un nombre variable de notes */

Remarques:
- Les tableaux à plusieurs dimensions sont définis à l'aide de tableaux des tableaux. Ce sont des tableaux dont les composantes sont elles-mêmes des tableaux. Par exemple Tab[][][]
- Un tableau à deux dimensions de tailles n et m, est en réalité un tableau unidimensionnel de n lignes, où chaque ligne est un tableau de m composantes.

Dimensions du tableau
Soit tab un tableau a deux dimensions:

- `tab.length`: donne la longueur de la première dimension, c'est-à-dire le nombre de lignes du tableau tab.

- `tab[i].length` : donne la longueur de la ligne i de tab, autrement dit, le nombre de colonnes de cette ligne.
-

Initialisation

Comme pour les tableaux à une dimension, on peut faire l'initialisation d'une matrice par énumération de ses composantes.

Exemple

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

/* créé une matrice à 3 lignes (un tableau à 3 composante). Chaque ligne (chaque composante) est un tableau unidimensionnel à un nombre variable de composantes: */

```
int [] t = tab[1]; // crée le tableau t= {5,6,8}
tab[1][2] = 8;
```

Librairies et packages

En Java les classes étaient organisées en *package* et en librairie

Librairie

Une *librairie* est un ensemble de classes Java qui offre un grand nombre de fonctionnalités par exemple des librairies pour manipuler des images, pour envoyer des e-mails, pour interagir avec le système d'exploitation, pour faire des animations 3D, ... ces librairies devront être disponible pour pouvoir compiler et exécuter le programme. Tout système Java possède une *librairie standard Java* qui contient les classe de base pour compiler et exécuter un programme. Les classes de cette librairie sont organisées en packages.

Package

Un package rassemble des classes et des interfaces autour d'une fonctionnalité précise et commune. Cette organisation en package est importante pour gérer l'*espace des noms* des classes. Deux classes du même package ne peuvent avoir le même nom, sinon il faut utiliser deux packages différents et pour accéder à une classe précise il faut rajouter le nom du package (comme par exemple `java.util.Date` et `java.sql.Date`). Le package `java.lang` est le package de base qui contient des classes qui seront presque toujours utilisées

Création de package

Pour définir un package, il suffit de commencer le fichier source (fichier .java) contenant les classes et interfaces à regrouper par l'instruction « `package nomPackage ;` ». L'identificateur nomPackage est le nom que l'on veut donner au package. Toutes les classes et interfaces contenues dans le fichier feront partie du même package.

Règles:

- Un package portant le nom "Package1" doit être stocké dans un répertoire de nom "Package1" que l'on place ensuite dans une arborescence de répertoires.
- Pour que le compilateur puisse trouver le package, il est essentiel qu'il connaisse l'emplacement du package. Pour cela Java utilise une variable d'environnement appelée **classpath** donnant la liste des chemins d'accès aux classes.
- Par défaut le compilateur (ainsi que la machine virtuelle) recherchent les classes dans le répertoire courant et le répertoire des classes spécifiés dans **classpath**.

Lorsque l'on désire utiliser une classe particulière, il faut normalement toujours utiliser son nom qualifié complet pour que le programme compile :

```
java.util.Date second = new java.util.Date (2012 - 1900, 1, 1);
```

Ce qui rend les choses un peu difficiles. Pour résoudre ce problème, on utilise le mot clé `import` suivi du nom qualifié complet de la classe :

Syntaxe

i*mport monPaquetage.maClasse ;* //utiliser 'maClasse' de 'monPaquetage'

Exemple

```
import java.util.Date ;

...........
Date second = new Date (2012 - 1900, 1, 1);
```

De plus, pour résoudre le problème de déclaration d'importation de plusieurs classe, il suffit d'importer le package les contenant (import java.util.*;).

Généralisation et héritage

Définition

Les concepts de généralisation et d'héritage sont fondamentaux en Java. Ils permettent d'écrire des programmes simplement et ils permettent la réutilisation de programmes. Le principe consiste àutiliser des classes existantes sans toucher à leurs codes internes tout en ajoutant de nouvelles méthodes et des nouveaux attributs pour personnaliser ces classes. Cela permet d'avoir plusieurs classes qui partage les mêmes ensembles d'attributs et des méthodes. La terminologie des super-classes et sous-classes est identique à celle de UML

Syntaxe:

class ClasseDerivee extends ClasseDeBase

- La classe ClasseDeBase est le nom de la classe de base. On l'appelle aussi ***classe mère***, ***classe parente*** ou ***super-classe***. Le mot clef ***extends*** indique la classe mère.
- La classe ClasseDerivee est le nom de la classe dérivée. Elle hérite de la classe ClasseDeBase. On l'appelle aussi ***classe fille*** ou ***sous-classe.***

Java ne permet pas l'héritage multiple. Il permet uniquement l'héritage ***simple***: chaque classe a une et une seule classe mère dont elle hérite les attribues et les méthodes. De plus Toutes les classes héritent implicitement de la classe « ***Object ».*** Par défaut, on ne met pas ***extends Object*** dans la définition d'une classe. Une classe déclarée ***final*** ne peut pas avoir de classes filles.

Lorsqu'une classe hérite d'une autre classe la classe dériver peut :

- bénéficier automatiquement des définitions, des attributs et des méthodes de la classe mère. Elle hérite de tous les attributs (y compris "static") et méthodes membres de la classe mère (sauf les méthodes privées et les constructeurs).
- avoir ses propres attributs et méthodes correspondants à sa sa propre spécificité
- Elle peut adapter les attributs et les méthodes de la classe existante :
  - redéfinir des méthodes (personnaliser les méthodes): exactement les mêmes noms des méthodes et les mêmes signatures.
  - surcharger des méthodes: les mêmes noms des méthodes mais pas les mêmes signatures (types et/ou arguments différents).

Exemple : considérons la classe suivante :

```
class Point {
      public int x;
      public int y;
```

```
      public Point(int a, int b) { x = a; y = b; }
      void deplace(int dx, int dy) { x += dx; y += dy; }
      public void affiche() { System.out.println( "x = " + x + "y = " +
y) ; }
}

class PointCol extends Point {
      public int couleur;
      public PointCol(int a, int b, int c) { super(a, b) ; colore(c); }
      public void colore(int c) { couleur = c }
}
```

Accès aux membres de la classe mère.

- - Un objet (une instance) d'une classe dérivée peut appeler depuis l'extérieur (accès directe depuis
- l'extérieur)
  - ses propres méthodes et attributs publics et protected.
  - les méthodes et les attributs publics et protected de la classe de base**.** La classe dérivée ne peut accéder directement (via l'opérateur *point* «.») qu'aux membres publics et protégés hérités.
- - Une méthode d'une classe dérivée a accès aux membres publics et protected de sa classe de base, mais elle n'a pas accès aux membres privées de sa classe de base.

Exemple

```
PointCol pc = new PointCol(1, 2, 3) ;
pc.deplace(4, 5) ;  // correct car 'deplace' est héritée
pc.colore(6) ;  // correct
Point p = new PointCol(7, 8) ;
p.colore(9) ;  // ERREUR ! l'héritage ne marche pas dans ce sens !
```

Construction des objets dérivés

Une instance de la classe dérivée est construite avec une partie provenant de la définition de la classe dérivée, et une autre provenant de la définition de la superclasse.
En Java, le constructeur de la classe drivée doit prendre en charge l'intégralité de la construction de l'objet (Le constructeur de la classe dérivée doit faire appel au constructeur de la classe de base en appelant explicitement la méthode ***super()*** ainsi que la liste des paramètres appropriés, l'instruction de l'appel doit être la première instruction dans la définition du constructeur de la classe dérivée)

Appel des constructeurs

- Si les classes de base et la classe dérivée ne définissent aucun constructeur alors c'est les constructeurs par défaut qui seront appelés (La création d'un objet dérivé s'effectue par appel du constructeur par défaut qui appelle implicitement le constructeur par défaut de la classe mère.
- Si la classe mère ne définit pas de constructeur et la classe dérivée en définit un, alors le constructeurs par défaut de la classe mère sera appelé implicitement.
- Si la classe mère ne définie que des constructeurs avec arguments, alors la classe dérivée doit définir au moins un constructeur. Dans chaque constructeur de la classe dérivée, un constructeur de la classe mère est explicitement appelé avec super() qui doit être dans la première instruction.
- Si la classe mère définie au moins un constructeur sans arguments, alors l'appel explicite avec ***super()*** n'est pas obligatoire. Cependant si le constructeur de la classe dérivée n'appelle pas explicitement le constructeur de la classe mère, alors c'est le constructeur sans arguments de la classe mère qui sera implicitement appelé.

- Si la classe mère définie au moins un constructeur sans arguments, et la classe dérivée n'en définit aucun constructeur, alors la création d'un objet dérivé à l'aide du constructeur par défaut qui fait appel au constructeur sans arguments de la classe mère.

Exemple :

```
class PointCol extends Point {
      public int couleur;
      public PointCol(int x, int y, int c) {
            super (x, y) ; // appel du constructeur de la super-classe
            couleur = c ;
      }.
..
}
```

Redéfinition de méthodes et des attributs

Méthode

Une méthode de la classe fille peut avoir la même signature qu'une fonction de la classe mère. On dit alors que la méthode de la super-classe est *redéfinie* dans la sous-classe.
Quand on est dans la sous-classe, on appelle la méthode de la sous-classe de façon habituelle. Si on veut appeler la méthode de la super-classe, il faut préciser le nom de la super-classe avec le mot-clé 'super' en préfixe
la redéfinition d'une méthode d'une classe consiste à fournir dans une sous-classe une nouvelle implémentation de la méthode. Cette nouvelle implémentation masque complètement celle de la superclasse.
Il ne faut pas confondre la ***redéfinition*** et la ***surcharge*** d'une méthode. Surcharger une méthode consiste à définir une nouvelle méthode avec le même nom, mais pas la même signature, qu'une autre méthode de la même classe

Exemple: Considérons la classe `Point` définie ci-dessus et

```
class PointCol extends Point { ...
      public void affiche() {
            super.affiche();
            System.out.println("couleur " + couleur);
      }
}
...
public static void main(String argv[]) {
      PointCol p = new PointCol(10, 20, 5);
      p.affiche();
      p.super.affiche();
      p.deplace(2,4);
      p.affiche();
      p.colore(2);
      p.affiche();
}
```

Attributs

```
class A { ... int a; char b; ... };
class B extends A {float a; ... };
```

Si `objetB` est de type `B`, `objetB.a` fait référence à l'attribut `'a'` de type `float` de la classe `B`. On accède à l'attribut `'a'` de type `int` par `objetB.super.a`. L'attribut `'a'` de la classe `B` s'ajoute à celui de la classe A

Compatibilité entre objets d'une super-classe et objets d'une sous-classe

Puisque la relation d'héritage est une relation forte on peut se demander si l'on peut affecter un objet de la super-classe dans un objet de la sous-classe ou pas et inversement.

Tout objet de la classe dérivée est un objet de la classe de base et en Java il existe des conversions implicites de type. On peut mettre un objet de type dérivé vers un type de base mais pas l'inverse.
Si A est la super-classe et B la sous-classe, on a :

```
A a;
B b;
```

alors `a = b` est ***légal*** alors que `b = a` est ***illégal***.
Par contre, très souvent, dans une sous-classe, une méthode de la super-classe retourne (une référence sur) un objet de la classe de base, mais puisque l'appel a été fait dans la sous-classe le programmeur sait que l'objet retourné est du type de la sous-classe donc il peut violer la règle avec l'opérateur de **cast** en écrivant : `b = (B) a ;`. Cependant cette opération peut produire une erreur si l'objet n'est pas de la classe dérivée B

L'opérateur « instanceof »

Pour vérifier si un objet appartient à une classe on utilise l'opérateur **instanceof.** Si l'instruction : `b instanceof B` ; retourne **`true`**. Alors cela signifie que b est une instance de B.
Si B est une sous classe de A alors l'instruction : `b instanceof A` ; retourne aussi **`true`**.
En particulier: (`b instanceof Object`) renvoie **`true`** car toutes les classes héritent de la classe Object.

Polymorphisme

Le Polymorphisme veut dire que le même service peut avoir un comportement différent suivant la classe dans laquelle il est utilisé. C'est un concept fondamental de la programmation objet, indispensable pour une utilisation efficace de l'héritage
Le polymorphisme est utilisé avec la redéfinition de méthode, ainsi, on parvient à obtenir des comportements différents selon l'objet actuellement référencé par la référence polymorphique
Exemple

```
class ClasseA {
      public void f(){
            System.out.println("Méthode f(): Classe de base A");
      }
      public void g(){
            System.out.println("Méthode g(): Classe de base A");
      }
}
class ClasseB extends ClasseA { // ClasseB hérite de la classe ClasseA
      public void f(){  // la méthode f() est redéfinie dans ClasseB
            System.out.println("Méthode f(): ClasseB dérivée de ClasseA");
      }
      public void h(){  // la méthode h() définie uniquement dans ClasseB
            System.out.println("Méthode h(): ClasseB dérivée de ClasseA");
      }
}
public class Polymorphisme {
      public static void main(String [] args ) {
            ClasseA a=new ClasseA();
            ClasseB b = new ClasseB();
            a.f();  // appelle la méthode définie dans ClasseA
            a=b;  // le type déclaré de a est ClasseA. Le type réel de a est ClasseB
            a.f();  // appelle la méthode définie dans ClasseB
            a.g();// appelle la méthode définie dans ClasseA
            b.g();// appelle la méthode définie dans ClasseA
            b.h():// appelle la méthode h() définie dans ClasseB
      }
```

```
}
```
Si les méthodes appelées sont

- ***usuelles*** (méthodes non `static`) : quand on manipule un objet via une référence à une classe mère, ce sont toujours les méthodes de la classe réelle (effective) de l'objet qui sont appelées.

Exemple

Soit f() une méthode non static redéfinie dans la classe `ClasseB .`

```
ClasseA objA = new ClasseB();
objA.f(); /* la méthode f() est redéfinie dans ClasseB qui est la classe réelle de l'objet, même
                    si objA est une référence de type ClasseA  classe de déclaration de l'objet. */
```

- ***de classe*** (méthodes `static`): quand on manipule un objet via une référence à une classe mère, ce sont toujours les méthodes de la classe de déclaration de l'objet qui sont appelées.

Exemple

Soit g() une méthode de classe (méthode static) redéfinie dans la classe « ClasseB ».

```
ClasseA objA = new ClasseB();
objA.g();  /*la méthode g() est définie dans ClasseA, classe de déclaration de l'objet,
             même si la classe réelle de objA est la classe ClasseB*/
```

En Java, dès la compilation, l'existence de la méthode appelée doit être garantie (typage statique) dans la classe de déclaration de l'objet ou dans une de ces classes ancêtres. Le polymorphisme est obtenu grâce au mécanisme de la liaison retardée (*late binding)*: la méthode qui sera exécutée est déterminée seulement à l'exécution, et pas dès la compilation par le type réel de l'objet qui reçoit le message (et pas par son type déclaré)

Exemple

Soient `ClasseB` une classe qui hérite de la classe `ClasseA`. Considérons les instructions suivantes :

```
ClasseA objB = new ClasseB();  /* la classe de déclaration de l'objet objB est ClasseA
                                  la classe réelle de l'objet objB est « ClasseB »*/
objB.f() ; // quelle méthode f()  sera appelée dans cette instruction.
```

Dans le cas où la méthode `f()` n'est pas définie ni dans la classe de déclaration (la classe `ClasseA`) de l'objet, `objB`, ni dans une de ses classe ancêtre alors erreur de compilation.

Dans le cas contraire, nous distinguons deux situation : si la méthode `f()` est redéfinie dans la classe `ClasseB` , alors c'est cette méthode qui sera appelée. Sinon, la recherche de la méthode `f()` se poursuit dans la classe mère de `ClasseB` , puis dans la classe mère de cette classe mère, et ainsi de suite, jusqu'à trouver la définition d'une méthode `f()` qui sera alors exécutée.

Classes abstraites et interfaces

Java propose un mécanisme pour traiter des classes ou méthodes abstraites.

Classes abstraites

Une classe abstraite est totalement l'opposé d'une classe final: c'est une classe qui ne peut pas être instanciée directement. Elle n'est utilisable qu'à travers sa descendance. Elle doit toujours être dérivée pour pouvoir générer des objets.

L'utilité d'une classe abstraite est de permettre le regroupement (la factorisation) des attributs et méthodes communs à un groupe de classes. Elle contient impérativement au moins méthodes abstraites.

Syntaxe

abstract class NomClasseAbstraite

Méthodes abstraites

Une méthode est dite abstraite si elle n'est pas implémentée (corps de méthode) mais juste déclarée. Cependant elle doit obligatoirement être implémentée différemment dans toutes les classes filles. L'intérêt des méthodes abstraites vient du fait que l'on peut les appeler de la même façon pour tous les objets dérivés (polymorphisme)
Une classe qui *possède une (ou plusieurs) méthode abstraite* est obligatoirement abstraite. Pour spécifier qu'une classe ou une méthode est abstraite on utilise le mot-clé '**abstract**'.

Syntaxe

abstract typeRetour NomMethodeAbstraite(type1,type2,...) ;

Interfaces

La structure syntaxique des classes Java ne permet pas une coupure naturelle entre les informations publiques et privées d'une classe : la définition des méthodes est accolée à leur déclaration (contrairement à C++ qui oblige le programmeur à placer la déclaration dans les *.h et les définitions dans les *.cpp). Cependant, Java prévoit des *interfaces* qui ne disposent que de méthodes abstraites et de données. On utilise le mot-clé '`interface`'.

Syntaxe

interface NomInterface

Pour les interfaces, il n'est pas nécessaire de rajouter le mot clé abstract devant les signatures des méthodes. De plus elle n'admettent aucun attribut mais peuvent posséder des constantes publics.
Les interfaces ne sont pas instanciable (par défaut abstraites) et pourront se dériver (une interface peut hériter d'une autre interface)
Une classe pourra alors implémenter une ou plusieurs interfaces en utilisant le mot-clé '`implements`'.

Syntaxe

class NomClasse implements NomInterface

Elle peut implémenter *complètement* l'interface en implémentant toute ses méthodes ou *partiellement* et dans ce cas doit être déclarée abstraite, Les méthodes manquantes devront être implémentées par ses classes filles
Un objet peut être déclaré avec une interface et créé avec une classe qui implémente cette interface.
Exemple :

```
interface NomInterface {
      public final int a = 0 ;
      void tourner(int) ;
}
class Classe1 implements NomInterface {
      void tourner(int x) {
            …
      }

}
class Classe1App{

      … … …
      NomInterface obj = new Classe1();
}
```

Gestion des exceptions

La gestion d'erreurs faites avec des paramètres de retour dans des fonctions est une technique

relativement lourde à gérer pour le programmeur. Pour cette raison java à introduit le mécanisme des exceptions.

Définition

Une exception est une erreur qui se produit pendant l'exécution d'un programme. Elle peut conduire soit à l'arrêt du programme soit à des résultats incorrects. Les exceptions peuvent être générées soit par l'environnement d'exécution Java, par exemple manque de place mémoire, soit à cause du non respect des règles du langage ou soit par le code (exceptions définies explicitement par les utilisateurs).

Le mécanisme des exceptions permet de :
- traiter les erreurs qui se produisent pendant l'exécution.
- séparer le code qui décrit le déroulement normal du programme et le code de gestion d'erreur.

Lorsqu'une exception survient, un objet représentant cette exception est créé dans la méthode qui a générée l'erreur. Cette exception est soit attrapée et traité dans cette méthode soit elle est passée pour être traité ailleurs.

Création d'une exception

Une méthode susceptible de lever une exception doit spécifier cette exception éventuelle à son appelant, en utilisant le mot-clé **throws** dans sa signature.
De plus,pour lancer l'exception, elle doit instancier un nouvel objet de cette exception en utilisant le mot clé **throw .**

Syntaxe

```
void maMethodeAvecExceptionPossible() throws monException {
        …
        throw monException;
}
```

monException doit être un type connu. Pour cela, on doit indiquer au minimum que **monException** est une extension de la classe pré-définie **Exception** :

Syntaxe

```
class monException extends Exception {}
```

Traitement d'une exception

L'appelant d'une méthode lançant des exceptions a le choix entre :

- ignorer l'exception et la transmettre au niveau appelant supérieur en utilisant à nouveau le mot-clé throws,
- récupérer l'exception avec des blocs de code qui utilisent les mot-clés try et catch.

Ignorer une exception

Dans ce cas l'exception est transmise au niveau appelant supérieur (comme une bulle) jusqu'à retrouver un bloc de traitement

Exemple

```
void maMethodeAppelanteSansRecupération() throws MonException {
        …       maMethodeAvecExceptionPossible() ;
        …
```

} // transmet au niveau supérieur

Attraper une exceptions

Il est possible d'attraper une exception, à l'aide du bloc **« try-catch »**pour faire un traitement relatif à ce problème. On peut donc personnaliser le message d'erreur et ensuite, soit arrêter le programme, soit faire un autre traitement

Règles :

- Le bloc « try » contient les instructions susceptibles de lever (déclencher, générer) une exception
- Le bloc « catch » sert de gestionnaire d'exception.
- Le paramètre de l'instruction « catch » indique le type de l'exception générée ainsi que le nom de l'instance de cette exception.
- On peut gérer plusieurs exceptions avec des instructions « catch » multiples. Dans ce cas l'environnement d'exécution cherche celle qui correspond à l'exception levée. La recherche s'effectue parmi les instructions « catch », les unes après les autres, dans l'ordre où elles sont écrites,

Syntaxe

```
void  maMethodeAppelanteAvecRecupération()   {
        try { // essai d'appeler la méthode
                maMethodeAvecExceptionPossible() ;
        }
        catch (MonException1 e1) { // récupère l'exception
                System.out.println(" Exception1 levee ") ;
                exit(0) ;
        }
        .. .. .. ..
        catch (MonExceptionN eN) { // récupère l'exception
                System.out.println(" ExceptionN levee ") ;
                exit(0) ;
        }
}
```

Dans cet exemple, `maMethodeAppelanteAvecRecupération` choisit de ne pas transmettre l'exception au niveau supérieur, elle récupère l'exception en affichant un message `ExceptionX levee` et en arrêtant le programme.

Attribut, instance d'exception

Dans le traitement précédent nous avons utilisé une classe exception, mais on peut aussi utiliser une *instance* d'une classe exception, définir des *attributs*, et même des méthodes dans la classe exception étendue :

Syntaxe

```
class   monException   extends Exception {
        public int monAttribut ;
}
```

Ainsi on peut lancer une instance d'exception :

```
{
```

```
        monException except = new monException() ;
        except.monAttribut = -1 ;
        throw except;
}
```

On peut alors traiter l'exception en utilisant la valeur de *monAttribut* de l'exception :

```
void maMethodeAppelanteAvecRecupération() {
        try {// essai d'appeler la méthode
                maMethodeAvecExceptionPossible() ;
        }
        catch (MonException e) { // récupère l'exception
                System.out.println(« Exception levee ») ;
                if (e.monAttribut==-1) exit(0) ;
        }
}
```