ELEMENTS DE BASE

DU LANGAGE JAVA

Préparé par Larbi Hassouni

Identificateurs Java

- Identificateurs
  - Sont des tokens qui représentent les noms des variables, méthodes, classes, etc.
  - Exemples d'identificateurs : taux, main, System, out.
- doit respecter les règles suivantes:

1. Il doit consister en une suite illimitée de caractères unicodes qui commence par une lettre, le signe $ ou le trait de soulignement (_).

   [a..z, A..Z, $, _]{a..z, A..Z, $, _, 0..9, Unicode}

2. Il ne doit pas être un mot clé, une valeur booléenne (true ou false) ou le mot réservé null

- Les identificateurs Java sont sensibles à la casse.
  - Cela signifie que les lettres majuscules et minuscules ne sont pas considérés identiques dans un identificateurs.

Par conséquent les identificateurs TAUX , Taux, taux ne sont pas identiques**.**

Conventions pour les identificateurs

Les noms des classes commencent par une majuscule . Si le nom est formé de plusieurs mots, la première lettre de chaque mot doit être en majuscule.

Exemple : **Object , System , Scanner , ExempleDeNomDeClasse**

Les variables et les méthodes commencent par une minuscule . Si le nom est formé de plusieurs mots, la première lettre de chaque mot doit être en majuscule à l'exception du 1er mot.

Exemple : **tauxTva, unEtudiant, exempleDeNomDeVariable**

Les noms des constantes sont en majuscules. Si le nom est formé de plusieurs mots, ils sont séparés par le caractère souligné « _ **» :**

Exemple : **TAILLE_ECRAN , EXEMPLE_DE_CONSTANTE**

Mots clés Java

● Les mots clés sont des identificateurs prédéfinis qui ont une signification spécifique en Java.
● Vous ne pouvez pas utiliser un mot clé comme nom de variable, classe, methode ... etc.
● La liste des mots clés est fournie ci-dessous.

abstract, boolean, break, byte, case, catch, char, class,

continue, default, do, double, else, extends, final, finally,

float, for, if, implements, import, instanceof, int, interface,

long, native, new, null, package, private, protected,

public, return, short, static, super, switch, synchronized,

this, throw, throws, transient, try, void, volatile, while

Commentaires Java

- Commentaires ?
  - Ce sont des notes écrites pour documenter le programme.
  - Ils n'ont aucun effet sur le déroulement du programme.
- Il existe 3 types de commentaires en Java
  - Commentaires sur une seule ligne (style C++)
    Comme en C++, ces commentaires commencent par un double stalsh (//), et sétendent sur une seule ligne.
    Exemples:
    //Commentaires sur toute la ligne
    int i; //Commentaire jusqu'à la fin de la ligne
  - Commentaires sur plusieurs lignes (style C)
    Comme en C, ces commentaires commencent par /* et se terminent par */. Ils peuvent s'étendre sur plusieurs lignes.
    Exemple:
    /*Ce type de commentaire peuvent s'étendre sur une ligne
    ou plusieurs lignes */

Commentaires Java

– Commentaires pour l'outil Javadoc, qui sont utilisés pour générer une documentation HTML sur le programme.

  Ces commentaires commencent par /** et se terminent par */

  Ils peuvent s'étendre sur plusieurs lignes.

  Ils peuvent contenir certaines balises pour ajouter davantage d'informations aux commentaires.

Exemple:

```
/**
    Ceci est un exemple de commentaires javadoc\n qui génére une
    documentation HTML. Il utilise les balises comme:
    @author MOHA ayoub
*/
```

Variables

Définition: Une variable est une donnée représentée par un identificateur

Vous devez explicitement donner un nom et un type à chaque variable que vous voulez utiliser dans le programme.

Le nom d'une variable doit être un identificateur autorisé.

Le nom d'une variable est utilisé pour référencer la donnée que la variable contient.

Le type d'une variable détermine quelles sont les valeurs que la variable peut contenir et quelles opérations peuvent être effectuées sur la variable.

Variables

Pour donner à une variable un nom et un type, nous écrivons une *déclaration* de variable de la manière suivante :

<type> <nomDeLaVariable>

Exemple : double tauxTva;

En plus du nom et du type, une variable à un **domaine de visibilité** et un **SCOPE**

La partie du programme où le nom d'une variable peut être utilisé par son simple nom (sans qualificatif) est appelé **scope** de la variable

Types de données (Data Types)

Chaque variable doit avoir un type de donnée.

Le type de donnée d'une variable détermine l'ensemble des valeurs que la variable peut contenir.

Il détermine également le type d'opérations qui peuvent être effectuées sur elle.

Par exemple:
Une variable de type **int** ne peut contenir que des valeurs entières (négatives ou positives).

Sur une variable entière nous pouvons effectuer des opérations arithmétiques (addition, soustraction …etc)

Types de données

Le langage Java contient deux catégories de types de données:

Types primitifs

Type référence

Une variable de type primitif contient une valeur simple (nombre, caractère, ou booléen).

Une variable de type référence contient la référence (ou adresse) d'un objet(voir classes).

Types de données

Types primitifs

Une variable de type primitif contient une valeur simple (nombre, caractère, ou booléen).

Cette valeur est stockée en mémoire sur un nombre de bits déterminé et selon un format bien précis.

Exemple:
Une variable de type **int** mémorise sa valeur sur 32 bits et utilise la représentation en complément à 2

Une variable de type **char** mémorise sa valeur sur 16 bits et utilise le format Unicode

A variable of primitive type contains a value of a particular size and format.

Plus grandes valeurs des types primitifs

```
public class MaxVariablesDemo
{
        public static void main(String args[])
        {
                //integers
                byte largestByte = Byte.MAX_VALUE;
                short largestShort = Short.MAX_VALUE;
                int largestInteger = Integer.MAX_VALUE;
                long largestLong = Long.MAX_VALUE;

                //real numbers
                float largestFloat = Float.MAX_VALUE;
                double largestDouble = Double.MAX_VALUE;

                //other primitive types
                char aChar = 'S';
                boolean aBoolean = true;
```

Plus grandes valeurs des types primitifs

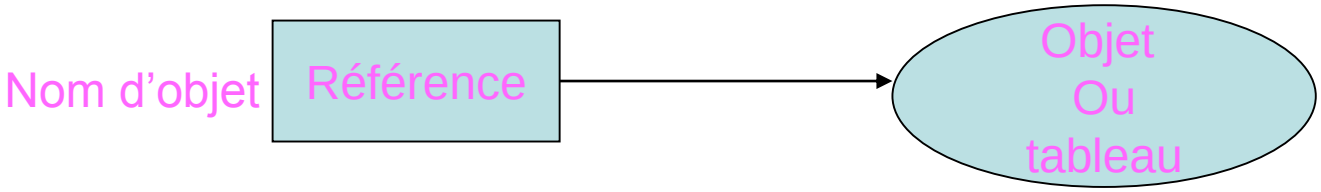
```
//Display them all.

System.out.println("The largest byte value is " + largestByte + ".");
System.out.println("The largest short value is " + largestShort + ".");
System.out.println("The largest integer value is " + largestInteger + ".");
System.out.println("The largest long value is " + largestLong + ".");
System.out.println("The largest float value is " + largestFloat + ".");
System.out.println("The largest double value is " + largestDouble + ".");
if (Character.isUpperCase(aChar))
 {
        System.out.println("The character " + aChar + " is uppercase."); }
else
{
        System.out.println("The character " + aChar + " is lowercase.");
}

System.out.println("The value of aBoolean is " + aBoolean + ".");

} //fin de main
} // fin de la classe
```

Types de données primitifs

MotClé	Description-Taille/Format	Intervalle
Entiers		
byte	**Entier signé sur 1octet en Cà2**	**-2^7 (-128) à + 2^7-1(127)**
short	**Entier signé sur 2 octets en Cà2**	**-2^{15}(-32768)à + 2^{15}-1**
int	**Entier signé sur 4 octets en Cà2**	**-2^{31} à + 2^{31}-1**
long	**Entier signé sur 8 octets en Cà2**	**-2^{63} à + 2^{63}-1**
Réels		
float	**Réel en virgule flottante simple précision sur 4 octets (IEEE 754)**	-3.4..e^{38} à -1.4..e^{-45} à 1.4..e^{-45} à 3.4..e^{38}
double	**Réel en virgule flottante double précision sur 8 octets (IEEE 754)**	**-1.7…e^{308} à -4.9…e^{-324} 4.9…e^{-324} à 1.7…e^{308}**
Autres types		
char	Un seul caractère sur 2 octets (Unicode)	Code unicode de 0 à 65535
boolean	Valeur boolèene sur 1 bit	true or false

Types Références

La valeur d'une variable de type référence est une référence (adresse de) à la valeur à ou à l'ensemble des valeurs représenté par la variable.



Une variable référence contient l'adresse d'un objet ou d'un tableau.

Une référence est appelé Pointeur ou adresse dans les autres langages(C, …)

Le Langage Java n'autorise pas l'utilisation explicite des pointeurs.

Les tableaux, Classes, et Interfaces sont des types Références.

Constantes ou Valeurs Litterals

Il est possible d'écrire directement une constante dans votre programme.

Par exemple, pour assigner la valeur 20 à une variable de type entier, vous pouvez écrire :

```
I = 20 ;
```

20 est une valeur littérale.

Le compilateur attribue a la valeur 20 le type int par défaut.

Le tableau de la page suivante présente différents type de valeurs littérales.

Constantes numériques

Une constante «entière» est de type **long si elle est** suffixée par «**L» et de type int sinon**

Une constante «flottante» est de type **float si elle est** suffixée par «**F» et de type double sinon**

Exemples

35 //Constante de type int

2589L // constante de type long

37.266D // constante de type double

4.567e2 // 456,7 de type double

.123587E-25F // de type float

Constantes de type caractère

Une constante de type char est un caractère Unicode entouré par 2 simples quotes ‘’

Exemples :
'A' 'a' 'ç' '1' '2'

\ caractère d’échappement pour introduire caractères spéciaux
'\t' tabulation
'\n' nouvelle ligne
'\r' retour chariot retour arrière
'\f' saut de page
…
'\\' '\‘' '\"'

’\u03a9’ ***(\u suivi du code hexadécimal à 4 chiffres d’un*** *caractère Unicode)*

'α'

Autres constantes

Constantes de Type booléen

false
true

Constante référence

null ***//***Référence inexistante
//(indique qu'une variable de type non primitif ne référence rien)

Constante de type String

String n'est pas un type primitif, c'est une classe, et donc un type référence
Une constante de type String est une suite de caractères délimités par deux guillemets(" ")

Exemple: " Ceci est une constante de type String "

Noms de variables

En Java, le nom d'une variable doit respecter les règles suivantes:

1. Il doit être un identificateur autorisé (voir identificateurs)
2. Il doit être unique dans son Scope.

   Une variable peut avoir le même nom que d'autres variables déclarées dans des Scopes différents.

Scope d'une variable

Le Scope d'une variable:

1. Définit la région du programme à partir de laquelle il est possible de référencer la variable par son simple nom
2. Détermine l'instant où le système réserve de la mémoire pour la variable et l'instant où il libère la mémoire occupée par la variable.

Le Scope est différent du domaine de visibilité.

Le domaine de visibilité s'applique uniquement aux variables membres et détermine si la variable peut ou non être utilisée en dehors de la classe dans laquelle elle a été déclarée. La visibilité est positionnée par un modificateur d'accès (Voir Classes)

Scope d'une variable

L'endroit de la déclaration d'une variable dans un programme établit son scope et la place dans une des quatre catégories suivantes:

1. Variable membre (member variable)
2. Variable locale (local variable)
3. Paramètre de méthode (method parameter)
4. Paramètre de gestionnaire d'exception (exception-handler parameter)

Variable membre (member variable)

Une variable membre est un membre d'une classe ou d'un objet.

Elle est déclarée à l'intérieur d'une classe mais en dehors de toute méthode ou constructeur.

Son scope est toute la classe dont elle est membre.

Nous étudierons en détail les variables membres dans une leçon ultérieure.

Variable locale (local variable)

Une variable locale est déclarée dans un bloc de code.

Un bloc de code est délimité par les accolades ({}).

Le Scope d'une variable locale s'étend de l'endroit où elle est déclarée jusqu'à la fin du bloc où elle est déclarée.

Toutes les variables déclarées dans la méthode **main** sont des variables locales. Le Scope de ces variables s'étend de leur lieu de déclaration jusqu'à la fin de la méthode main indiquée par l'accolade fermante }.

Paramètre de méthode (method parameter)

Les paramètres de méthodes sont les argument formels des méthodes et constructeurs.

Les paramètres sont utilisés pour passer des valeurs aux méthodes et aux constructeurs.

Le Scope d'un paramètre est la méthode (ou constructeur) entière pour laquelle il est défini.

Paramètres de gestionnaire d'exception (exception-handler parameter)

Les paramètre de gestionnaire d'exception sont similaires aux paramètres de méthode mais sont paramètres d'un gestionnaire d'exception et non d'une méthode ou d'un constructeur.

Le Scope d'un paramètre de gestionnaire d'exception est le bloc de code encadré par les accolades {} qui vient après l'instruction **catch**.

Nous étudierons en détails ces paramètres dans la section : **Gestion des erreurs à l'aide des exceptions.**

Exemples de variables locales

Considérons le code ci-dessous:

```
if (...)
{
        int i = 17; ...
}
System.out.println(« La valeur de  i = " + i); // erreur
```

La dernière ligne de ce programme générera une erreur de compilation parce que la variable locale **i** est utilisé en dehors de son Scope.

Le Scope de la variable **i** est le bloc de code compris entre **{** et **}**.
La variable **i** n'existe plus après l'accolade fermante **}**.

Pour remédier au problème:
-il faut soit déplacer la déclaration de la variable **i** en dehors du bloc de code de l'instruction **if**,

-ou déplacer l'appel de la méthode **println** à l'intérieur du bloc du code de l'instruction **if**.

Initialisation de variable

Une variable locale ou variable membre peut être initialisée par une instruction d'affectation lors de sa déclaration.

Le type de la variable doit être compatible avec le type de la valeur qui lui est affectée.

L'exemple ci-dessous montre des exemples d'initialisation de variables:

```
//Entiers
byte largestByte = Byte.MAX_VALUE;
short largestShort = Short.MAX_VALUE;
int largestInteger = Integer.MAX_VALUE;
long largestLong = Long.MAX_VALUE;
//Nombres réels
float largestFloat = Float.MAX_VALUE;
double largestDouble = Double.MAX_VALUE;
//Autres types primitifs
char aChar = 'S';
boolean aBoolean = true;
```

Les paramètres des méthodes et des gestionnaires d'exception ne sont pas initialisées selon cette méthode; Ils reçoivent leur valeur à partir du code appelant

Variables « Finales » (Final Variables)

On peut déclarer une variable de n'importe quel Scope comme étant **final.**

La valeur d'une variable **final** ne peut pas être modifié après son initialisation.

Une variable final est similaire à une constante en C, C++ et C#.

Pour déclarer une variable final, utilisez le mot clé final dans la déclaration de la variable juste avant le type.

Exemple:

final int aFinalVariable = 0

Cette instruction déclare une variable final et l'initialise une fois pour toute.

Si on essaie de lui assigner une autre valeur dans la suite du programme, le compilateur engendrera une erreur.

Variables « Finales » (Final Variables)

Il est possible de déférer l'initialisation d'une variable locale. Il faut tout simplement déclarer la variable comme étant final, puis l'initialiser par la suite. Exemple:

```
final int aBlankFinalVariable;
……….
aBlankFinalVariable = 0
```

Une variable locale final qui a été déclarée sans être initialisée est appelé variable « blank final ». Une fois cette variable est initialisée, toute tentative future pour lui assigner une autre valeur engendrera une erreur lors de la compilation.

Forcer le type en java
Opérateur Cast

Java est un langage fortement typé

Dans certains cas, il est nécessaire de forcer le programme à considérer une expression comme étant d'un type qui n'est pas son type réel ou déclaré

On utilise le *cast :* ***(type-forcé) expression***

Exemple:

```
int i = 64;
char c = (char)i;
```

Cast entre type primitifs

Un ***cast*** *entre types primitifs peut occasionner une perte de* données

Par exemple, la conversion d'un ***int*** *vers un* ***short*** *peut donner un nombre complètement différent du nombre de départ*

Un ***cast*** *peut provoquer une simple perte de précision*

Par exemple, la conversion d'un ***long*** *vers un* ***float*** *peut faire perdre des chiffres significatifs mais pas l'ordre de grandeur*

Cast entre type primitifs

Les affectations entre types primitifs peuvent utiliser un *cast* implicite si elles ne peuvent provoquer qu'une perte de précision (ou, encore mieux, aucune perte)

```
int i = 130;
double x = 20 * i;
```

Sinon, elles doivent comporter un *cast explicite*

```
short  s = 65; // cas particulier affectation int "petit"

s = 1 000 000; // provoque une erreur de compilation

Int  i = 130;

byte b = (byte)(i + 2);

char c = (char)i; // caractère dont le code est 65

b = (byte)i; // b = -126 !
```

Cast entre entiers et caractères

La correspondance **char → int, long** s'obtient par *cast* implicite
Les correspondances

char → short, byte
et
long, int, short ou byte → char

nécessitent un *cast* explicite (les entiers sont signés et pas les **char)**

```
int i = 80;

char c = 68;          // caractère dont le code est 68

c = (char)i

i = c;

short s = (short)i;

char c2 = s;          // provoque une erreur
```

Opérateurs

Un opérateur effectue une opération (ou fonction) sur un, deux ou trois

opérandes.

Un opérateur qui requiert un seul opérande est appelé **opérateur unaire**.

Par exemple, ++ est un opérateur unaire qui incrémente la valeur de son opérande de 1.

Un opérateur qui requiert deux opérandes est un **opérateur binaire**.

Par exemple, = est un opérateur binaire qui affecte la valeur de son opérande qui se trouve à sa droite à son opérande suit se trouve à sa gauche.

En dernier, un opérateur qui requiert trois opérandes est appelé opérateur ternaire. Il existe un seul opérateur ternaire qui est ?:

C'est un raccourci de l'instruction if-else.

Exemple : Max = a>b?a:b ⇔ (si (a>b) Max = a else Max=b)

Opérateurs

Les opérateurs unaires supportent les deux notations **prefix** et **postfix.**

La notation **prefix** signifie que l'opérateur apparaît avant son opérande.

Opérateur op *// notation prefix (op signifie opérande).*

La notation postfix signifie que l'opérateur apparaît après son opérande

Op Opérateur *//notation postfix*

Tous les opérateurs binaires utilisent la notation **infix,** qui signifie que l'opérateur apparaît entre ses opérandes.

Op1 opérateur op2 *//notation infix*

L'opérateur ternaire utilise aussi la notation infix, car chaque composante de l'opérateur apparaît entre deux opérandes.

Op1 ? Op2 : Op3 */notation infix*

Opérateurs

En plus d'effectuer une opération, un opérateur retourne une valeur.

La valeur retournée et son type dépendent de l'opérateur et du type de ses opérandes.

Par exemple, les opérateurs arithmétiques tels que les opérateurs d'addition et de soustraction retournent comme résultat un nombre dont le type dépend de type de leurs opérandes.

Si on additionne deux entiers , le résultat retourné est un entier,

si on additionne deux réels le résultat retourné est un réel.

Opérateurs

Les opérateurs se répartissent en les catégories suivantes :

- Opérateurs arithmétiques
- Opérateurs relationnels et logiques
- Opérateurs de décalage et bit à bit
- Opérateurs d’affectation
- Autres opérateurs

Opérateurs arithmétiques

Le lanagage Java fournit un ensemble d'opérateurs arithmétique qui opèrent sur des nombres entiers et en virgules flottante.

Ces opérateurs sont : + (addition), -(soustraction), *(multiplication), /(division), et %(modulo). Ils ont la même signification qu'en C.

Le tableau suivant présente ces opérateurs:

Opérateurs arithmétiques binaires

Operateur	Format	Description
+	op1 + op2	Ajoute op1 à op2; il est aussi utilisé également pour concaténer deux Strings.
-	op1 - op2	Soustrait op2 de op1.
*	op1 * op2	Multiplie op1 par op2.
/	op1 / op2	Divise op1 par op2.
%	op1 % op2	Calcule le reste de la division de op1 par op2.

Types des résultats des Opérations Arithmétiques

Noter que lorsqu'un entier et un nombre en virgule flottante sont utilisés ensemble dans une opération arithmétique, le résultat est un nombre en virgule flottante. Le nombre entier est implicitement converti en un nombre en virgule flottante avant que l'opération ne soit effectuée.

Le tableau suivant présente en résumé les types de données retournés par les opérateurs arithmétiques en fonction des types des opérandes utilisés. Les conversions nécessaires sont effectuées implicitement avant que l'opération arithmétique ne soit effectuée.

Types des résultats des Operations Arithmétiques

Type du Resultat	**Type des Opérandes**
long	Aucun opérande n'est de type float ou double (Arithmétique entière); au moins un opérande est de type long.
int	Aucun opérande n'est de type float ou double (Arithmétique entière); aucun opérande n'est de type long.
double	Au moins un opérande est de type double.
float	Au moins un opérande est de type float; aucun opérande n'est de type double.

Opérateurs Arithmétiques Unaires

En plus des formes binaire des opérateurs + et -,

chacun de ces opérateurs possède une forme unaire, comme le montre le tableau suivant.

Operateurs Arithmétiques Unaires

Operateur	Utilisation	Description
+	+op	Promouvoit op au type **int** s'il est de type **byte, short,** ou **char**
-	-op	Calcule l'opposé arithmétique de op

Opérateurs d'incrémentation (++)
Et de décrémentation (--)

Il existe deux opérateurs arithmétiques considérés comme des raccourcis.

Le premier est l'opérateur ++ qui incrémente son opérande de 1.

Le deuxième est l'opérateur -- qui décrémente son opérande de 1.

Chacun de ces deux opérateurs peut apparaître avant (prefix) ou après (postfix) son opérande.

La version prefix , ++op/--op, fournit comme résultat la valeur de l'opérande après l'avoir incrémenté/décrémenté.

La version postfix op++/op--, fournit comme résultat la valeur de l'opérande avant de l'incrémenter/décrémenter.

Récapitulatif sur les opérateurs d'incrémentation(++) et de décrémentation (--)

Operateur	Utilisation	Description
++	op++	Incrémente op par 1; évalue l'expression en la valeur de op avant qu'il ne soit incrémenté.
++	++op	Incrémente op par 1; évalue l'expression en la valeur de op après qu'il soit incrémenté.
--	op--	Décrémente op par 1; évalue l'expression en la valeur de op avant qu'il ne soit décrémenté.
--	--op	Décrémente op par 1; évalue l'expression en la valeur de op après qu'il soit Décrémenté.

Operateurs Relationnels et Logiques

Un opérateur relationnel permet de comparer deux valeurs. Il retourne toujours une valeur boolèenne.

Par exemple l'opérateur **!=** retourne la valeur booléenne true si les deux opérandes ne sont pas égaux.

Le tableau ci-dessous fournit les différents opérateurs relationnels.

Operateurs Relationnels

Operateur	Utilisation	Description
>	op1 > op2	Retourne true si op1 est supérieur à op2
>=	op1 >= op2	Retourne true si op1 est supérieur ou égale à op2
<	op1 < op2	Retourne true si op1 est inférieur à op2
<=	op1 <= op2	Retourne true si op1 est inférieur ou égale à op2
==	op1 == op2	Retourne true si op1 est égale à op2
!=	op1 != op2	Retourne true si op1 est différent de op2

Operateurs Logiques (ou conditionnels)

Les opérateurs relationnels sont souvent utilisés avec les opérateurs logiques pour former des expressions conditionnelles plus complexes.

Le langage Java fournit six opérateurs logiques, 5 sont binaires et un est unaire

Le tableau suivant présente ces six opérateurs.

Operateurs Logiques (ou conditionnels)

Opérateur	Utilisation	Description
&&	op1 && op2	Retourne true si op1 et op2 sont tous les deux true; evalue op2 uniquement dans le cas ou op1 est true
||	op1 || op2	Retourne true si op1 ou op2 est true; evalue op2 uniquement dans le cas où op2 est false.
!	!op	Retourne true si op est false
&	op1 & op2	Retourne true si op1 et op2 sont tous les deux booléens et ont la valeur true; il evalue toujours op1 et op2; Si les deux opérandes sont des nombres, il effectue l'opération AND bit à bit.
|	op1 | op2	Retourne true si op1 et op2 sont tous les deux booléens et op1 ou op2 est true; evalue toujours op1 et op2; si les deux opérandes sont des nombres, il effectue l'opération OR inclusif bit à bit.
^	op1 ^ op2	Retourne true si op1 et op2 sont differents — c'est-à-dire, si un seul opérande, et non les deux est true.

Opérateurs de décalage et bit à bit?????

Un opérateur de décalage effectue une manipulation des bits en effectuant un décalage gauche ou droite des bits de son opérande de gauche. Le tableau Ci-dessous présente les opérateurs de décalage fourni par le langage Java.

Operateurs de décalage

Operateur	Utilisation	Description
<<	op1 << op2	Effectue un décalage gauche des bits de op1 un nombre de fois spécifié par l'opérande op2; remplit avec 0 les bits de droite
>>	op1 >> op2	Effectue un décalage droite des bits de op1 un nombre de fois égal à op2. Remplit avec le bit le plus significatif (bit de signe) les bits de gauche.
>>>	op1 >>> op2	Effectue un décalage droite des bits de op1 un nombre de fois égal à op2. Remplit avec 0 les bits de gauche.

Operateurs Logiques

Le tableau ci-dessous présente les quatre opérateurs logiques fourni par Java pour Effectuer des opérations bit à bit sur leurs opérandes

Operateurs Logiques

Operateur	Utilisation	Operation
&	op1 & op2	-Opération AND bit à bit si les deux opérandes sont des nombres; -Opération AND logique si les deux opérandes sont booléen
|	op1 | op2	-Opération OR bit à bit si les deux opérandes sont des nombres; -Opération OR logiques si les deux opérandes sont booléens
^	op1 ^ op2	Opération OR exclusif(XOR)
~	~op	Complément à un des bits de l'opérande op

Opérateurs d'affectation

Le langage Java utilise l'opérateur = pour affecter une valeur à une variable.

Le langage Java fournit aussi un ensemble d'opérateurs qui combine une opération arithmétique, de décalage, ou de manipulation de bits avec l'opération d'affectation.

Supposez que vous voulez ajouter un nombre à une variable, et puis affecter le résultat à la variable en question, comme suit:

i = i + 2;

Vous pouvez écrire de façon plus concise cette instruction en utilisant l'opérateur +=.

Les deux instructions **i = i + 2;** et **i += 2;** sont équivalentes

Operateurs d'affectation

Le tableau suivant présente les différents opérateurs d'affectation fournis par Java.

Operateur	Utilisation	Description
+=	op1 += op2	Equivalent à op1 = op1 + op2
-=	op1 -= op2	Equivalent à op1 = op1 - op2
*=	op1 *= op2	Equivalent à op1 = op1 * op2
/=	op1 /= op2	Equivalent à op1 = op1 / op2
%=	op1 %= op2	Equivalent à op1 = op1 % op2
&=	op1 &= op2	Equivalent à op1 = op1 & op2
|=	op1 |= op2	Equivalent à op1 = op1 | op2
^=	op1 ^= op2	Equivalent à op1 = op1 ^ op2
<<=	op1 <<= op2	Equivalent à op1 = op1 << op2
>>=	op1 >>= op2	Equivalent à op1 = op1 >> op2
>>>=	op1 >>>= op2	Equivalent à op1 = op1 >>> op2

Autres Opérateurs

Le langage Java fourni d'autres opérateurs qui sont fournis dans le tableau ci-dessous.

Operateur	Utilisation	Description
?:	op1 ? op2 : op3	si op1 est true, retourne op2; sinon, reoturne op3
[ ]	Voir Création et utilisation des tableaux	Utilisé pour déclarer un tableau, pour créer un tableau, et pour accéder aux éléments d'un tableau.
.	Voir Utilisation des Objets	Utilisé pour former des noms longs (ou nom qualifiés)
(*params*)	Voir Définition des méthodes	Delimite une liste de paramètres séparés par des virgules
(*type*)	(*type*) op	Convertit op vers le type spécifié; une exception est relevée si le type de op n'est pas compatible avec type
new	Voir Utilisation des Objets et Création et Utilisation des tableaux	Crée un nouveau objet ou tableau.
instanceof	op1 instanceof op2	Retourne true si op1 est une instance de op2.

Operateurs Java

Le tableau ci-dessous donne la liste complète des opérateurs du langage Java.

Operateur	Utilisation	Description
+	+op	Promotes op to int if it's a byte, short, or char
-	-op	Arithmetically negates op
+	op1 + op2	Adds op1 and op2; also used to concatenate strings
-	op1 - op2	Subtracts op2 from op1
*	op1 * op2	Multiplies op1 by op2
/	op1 / op2	Divides op1 by op2
%	op1 % op2	Computes the remainder of dividing op1 by op2
++	op++	Increments op by 1; evaluates to the value of op before it was incremented
++	++op	Increments op by 1; evaluates to the value of op after it was incremented
--	op--	Decrements op by 1; evaluates to the value of op before it was decremented
--	--op	Decrements op by 1; evaluates to the value of op after it was decremented

>	op1 > op2	Returns true if op1 is greater than op2
>=	op1 >= op2	Returns true if op1 is greater than or equal to op2
<	op1 < op2	Returns true if op1 is less than op2
<=	op1 <= op2	Returns true if op1 is less than or equal to op2
==	op1 == op2	Returns true if op1 and op2 are equal
!=	op1 != op2	Returns true if op1 and op2 are not equal
&&	op1 && op2	Returns true if op1 and op2 are both true; conditionally evaluates op2
||	op1 || op2	Returns true if either op1 or op2 is true; conditionally evaluates op2
!	!op	Returns true if op is false
&	op1 & op2	Returns true if op1 and op2 are both boolean and both true; always evaluates op1 and op2; if both operands are numbers, performs bitwise AND operation
|	op1 | op2	Returns true if both op1 and op2 are boolean and either op1 or op2 is true; always evaluates op1 and op2; if both operands are numbers, performs bitwise inclusive OR operation
^	op1 ^ op2	Returns true if op1 and op2 are different — that is, if one or the other of the operands, but not both, is true

<<	op1 **<<** op2	Shifts bits of op1 left by distance op2; fills with 0 bits on the right side
>>	op1 **>>** op2	Shifts bits of op1 right by distance op2; fills with highest (sign) bit on the left side
>>>	op1 **>>>** op2	Shifts bits of op1 right by distance op2; fills with 0 bits on the left side
&	op1 & op2	Bitwise AND if both operands are numbers; conditional AND if both operands are boolean
|	op1 | op2	Bitwise OR if both operands are numbers; conditional OR if both operands are boolean
^	op1 ^ op2	Bitwise exclusive OR (XOR)
~	~op	Bitwise complement

=	op1 = op2	Assigns the value of op2 to op1
+=	op1 += op2	Equivalent to op1 = op1 + op2
-=	op1 -= op2	Equivalent to op1 = op1 - op2
*=	op1 *= op2	Equivalent to op1 = op1 * op2
%=	op1 %= op2	Equivalent to op1 = op1 % op2
&=	op1 &= op2	Equivalent to op1 = op1 & op2
|=	op1 |= op2	Equivalent to op1 = op1 | op2
^=	op1 ^= op2	Equivalent to op1 = op1 ^ op2
<<=	op1 <<= op2	Equivalent to op1 = op1 << op2
>>=	op1 >>= op2	Equivalent to op1 = op1 >> op2
>>>=	op1 >>>= op2	Equivalent to op1 = op1 >>> op2

?:	op1 ? op2 : op3	If op1 is true, returns op2; otherwise, returns op3
[ ]	Voir Création et utilisation des tableaux	Used to declare arrays, to create arrays, and to access array elements
.	Voir utilisation des objets	Used to form long names
(*params*)	Voir Définition des méthodes	Delimits a comma-separated list of parameters
(*type*)	(*type*) op	Casts (converts) op to the specified type; an exception is thrown if the type of op is incompatible with *type*
new	Voir utilisation des objets et création et utilisation destableaux	Creates a new object or array
instanceof	op1 instanceof op2	Returns true if op1 is an instance of op2

Expression, Instructions et Blocs

Expressions

Une *expression* est une combinaison de variables, d'opérateurs, et d'appels de méthodes, qui est construite en respectant la syntaxe du langage , et dont l'évaluation produit une seule valeur.

Les expressions sont utilisées pour calculer et affecter des valeurs aux variables. Elles sont aussi utilisées pour aider au contrôle du déroulement de l'exécution d'un programme

Ordre de priorité des opérateurs

Soit l'expression:

4 + 7 * 2

Quel est le résultat de son évaluation? :

11 * 2 = 22 //évaluation du + en premier

ou

4 + 14 = 18 // évaluation de * en premier

Pour éviter cette ambiguité on peut utiliser les parenthèses: Par exemple :

(4 + 7) * 2 : donnera comme résultat 22 car l'expression entre parenthèse est évaluée en premier

En absence de parenthèses l'ordre d'évaluation est déterminé par l'ordre de priorité des opérateurs qui est établi par le langage Java et fourni dans le tableau suivant

Ordre de Priorité des opérateurs dans l'ordre décroissant

Type d'opérateurs	Priorité
postfix	*expr*++ *expr*--
unaire	++*expr* --*expr* +*expr* -*expr* ~ !
multiplicatif	* / %
additif	+ -
décalage	<< >> >>>
relationnel	< > <= >= instanceof
egalité	== !=
AND bit à bit	&
XOR bit à bit	^
OR bit à bit	|
AND logique	&&
OR logique	||
Conditionnel	? :
Affectation	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Ordre de Priorité des opérateurs

Les opérateurs qui ont une priorité plus élevée s'évaluent avant ceux qui ont une priorité relativement plus faible.

Donc l'expression 4 + 7 * 2 est équivalente à 4 + (7 * 2) = 18

Les opérateurs qui se trouvent sur la même ligne ont la même priorité.

Lorsqu'une expression contient des opérateurs de même priorité, les opérateurs binaires, excepté les opérateurs d'affectation, sont évalués de la gauche vers la droite ;

Exemple : 14 / 2 * 3 = 7 * 3 = 21

par contre les opérateurs d'affectation sont évalués de la droite vers la gauche.

Exemple: y = x = 5; est équivalente à y = (x = 5)

Fonctions Mathématiques

Les fonctions mathématiques classiques se trouvent dans la classe Math. Le tableau ci-dessous fournit quelques fonctions:

Fonction	Description
sqrt	double sqrt(double x) : Calcule la racine carrée d'un réel positif x
pow	double pow(double x, double a): Calcule la puissance x^a.
sin	double sin(double x): calcule le sinus de x
cos	double cos(x) : calcule le cosinus de x
exp	double exp(double x): Calcule l'exponentielle de x(e^x)
log	double log(double x):Calcule le logarithme de x
round	long double round(x) : retourne l'entier le plus proche de x

Remarque: Toutes ces méthodes sont statiques et pour les utiliser directement sans préfix, il faut ajouter en haut de votre programme la ligne:

```
import static java.lang.Math.*;
```