Algorithmie et Programmation

Chapitre 1 : Types, affectation et lecture/écriture

1. Variables et typages.

1.1.Déclaration.

Dans un programme informatique, on va avoir en permanence besoin de stocker provisoirement des valeurs. Il peut s'agir de données issues d'un fichier sur disque dur, fournies par l'utilisateur (saisies au clavier), ou extraites d'une base de données. Il peut aussi s'agir de résultats obtenus au cours de l'exécution du programme. Ces données sont stockées en mémoire et ont une durée de vie égale au temps d'exécution du programme. Elles peuvent être de plusieurs types : des nombres, du texte, etc. Dès que l'on a besoin de stocker une valeur dans un programme, on utilise une variable.

Pour employer une image, une variable est une boîte, repérée par une étiquette. Pour avoir accès au contenu de la boîte, il suffit de la désigner par son étiquette.

En réalité, pour un ordinateur, ces variables sont stockées dans la mémoire vive et elles sont désignées par des adresses binaires. Il est beaucoup plus facile d'employer des étiquettes de son choix, que de devoir manipuler des adresses binaires.
Ainsi, la première chose à faire avant de pouvoir utiliser une variable est de définir « la boîte » et de lui donner une étiquette. Ceci se fait tout au début de l'algorithme, avant même les instructions proprement dites. C'est ce qu'on appelle la déclaration des variables.

Le nom de la variable (l'étiquette de la boîte) obéit à des impératifs changeant selon les langages. Toutefois, une règle absolue est qu'ils peuvent comporter des lettres et des chiffres, mais qu'ils excluent tous les signes de ponctuation, en particulier les espaces. Un nom de variable correct commence impérativement par une lettre minuscule.
Si pour des raisons purement pratiques on évite généralement les noms à rallonge, il faut tout de même veiller à choisir un nom pertinent en rapport à ce que représente la variable. Ainsi, on évitera d'utiliser de simples lettres souvent utilisées en mathématiques comme `x`, `y`, `i`, `j` …, qui sont peu explicites. On préférera des noms comme : `compteur`, `temperature`, `abscisse`, `ordonnee`, `point`, `somme`, `moyenne` …
Des noms explicites apportent du sens, et rendent tout simplement les algorithmes plus lisibles et compréhensibles.
Quant au nombre maximal de signes pour un nom de variable, il dépend du langage utilisé.

1.2.Type des variables.

En déclarant une variable, on souhaite réserver un emplacement mémoire ; mais cette déclaration doit être précisée. En effet, la taille de la variable et le type de codage utilisé dépendent de ce que l'on souhaite y stocker. On parle alors de typage. En typant une variable, on précise si celle-ci représente un entier, un réel, un caractère, etc...

Pourquoi typer ? Cela permet au compilateur de contrôler les affectations et ainsi d'éviter beaucoup d'erreurs et surtout cela lui permet de savoir quelle place réserver en mèmoire.

1.2.1.Types numériques classiques.

Commençons par le cas le plus fréquent, celui d'une variable destinée à recevoir des nombres. Si l'on réserve un octet (8 bits) pour coder un nombre, on ne pourra coder que

2^8 = 256 valeurs différentes. Cela peut signifier par exemple les nombres entiers de 1 à 256, ou de 0 à 255, ou de -127 à +128, c'est une pure question de convention.
Mais ce qui n'est pas une convention, c'est qu'on est limité à 256 valeurs possibles. Si l'on réserve deux octets (16 bits), on a droit à 65 536 valeurs ; avec trois octets(24 bits), 16 777 216, etc.

Le codage (type de variable) choisi pour un nombre va déterminer :
- les valeurs maximales et minimales des nombres pouvant être stockés
- la précision de ces nombres (dans le cas de nombres décimaux).

Tous les langages offrent un « bouquet » de types numériques, dont le détail est susceptible de varier légèrement d'un langage à l'autre. Généralement, on retrouve deux catégories de types numériques, les entiers et les réels. Le language JAVA dispose des types numériques suivants :

	Type	Taille octets	Description
Entiers	byte	1	Valeurs entières (entier signé): [-128 ; +127]
	short	2	Valeurs entières (entier signé) : [-32,768 ; +32,767]
	int	4	Valeurs entières (entier signé): [-2,147,483,648 ; 2,147,483,647]
	long	8	Valeurs entières (entier signé) : [-9,223,372,036,854,775,808 ; 9,223,372,036,854,775,807]
Réels	float	4	Nombre réel en virgule flottante [-3.402823E38 ; -1.401298E-45] pour les valeurs négatives [1.401298E-45 ; 3.402823E38] pour les valeurs positives
	double	8	Nombre réel en virgule flottante [-1.79769313486231E308 ; -4.94065645841247E-324] et [4.94065645841247E-324 ; 1.79769313486232E308]

Pourquoi ne pas déclarer toutes les variables numériques en réel double ? En vertu du principe de l'économie de moyens. Un bon algorithme ne se contente pas de fonctionner ; il le fait en évitant de gaspiller les ressources de la machine. Sur certains programmes de grande taille, l'abus de variables surdimensionnées peut entraîner des ralentissements notables à l'exécution (à cause d'une surchage de la mémoire entre autre). Alors, autant prendre de bonnes habitudes. S'il est facile de distinguer au moins les entiers des réels dans nos algorithmes, savoir si nous devons utiliser des entiers simples ou longs est encore mieux.

Exemple d'une section de déclaration des variables :

```
byte age;
double rayon;
```

1.2.2.Type caractère.

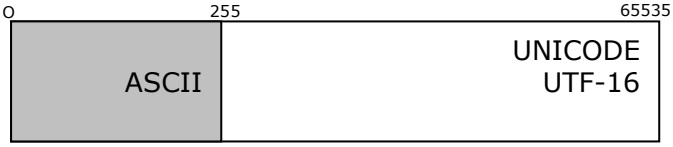
Tout langage de programmation propose un type alphanumérique pour stocker un caractère.

Une variable de type **char** permet de stocker un seul caractère, qu'il s'agisse d'une lettre (en minuscule ou en majuscule), d'un signe de ponctuation, d'un espace, d'un chiffre... L'ensemble des caractères disponible est défini par une table des caractères. Java utilise la table Unicode,

En effet, puisqu'un ordinateur ne sait finalement que manipuler des valeurs binaires, cette table associe un code binaire à chaque caractère. Le fait de déclarer le type **char** pour une variable indique que celle-ci doit être considérée au regard de cette table d'association. Selon le type de codage, la valeur 65 (codée en binaire en mémoire) correspond à la lettre **A** pour le type **char.**

La table Unicode est une table dont le jeu de caractères permet notamment de coder tous les caractères de toutes les langues (cf. http://www.unicode.org, http://www.ramou.net/zhUnicode.htm).

Remarque : la table ASCII est un sous-ensemble de la table Unicode.



La table Unicode sera notre table de référence pour ce cours. En effet, en Java, la taille d'un type **char** est de deux octets (soit 16bits, correspondant à l'Unicode UTF-16).

Exemple de déclaration d'une variable de type **char**:

```
char lettre;
```

1.2.3.Type booléen.

Le type booléen (boolean en Java) permet de stocker uniquement les valeurs logiques **true** et **false**.

Exemple de déclaration d'une variable de type **boolean**:

```
boolean estMajeur;
```

2. Affectation, expressions et opérateurs.

2.1.L'instruction d'affectation.

Après la déclaration d'une variable et pour pouvoir l'utiliser, il faut tout d'abord lui attribuer une valeur. Pour cela, on utilise l'instruction d'affectation. Cette instruction se note avec le signe **=**. Ce signe n'a rien à voir avec son équivalent en mathématiques.

- Affectation simple : type numérique

```
short unAge;
unAge = 27;
```

La variable unAge est d'abord déclarée de type **short**, puis on lui affecte la valeur **27**.

```
short ageJulien;
short ageOlivier;
ageOlivier = 27;
ageJulien = ageOlivier;
```

On peut attribuer à une variable la valeur d'une autre variable. La valeur de `ageJulien` est maintenant celle de `ageOlivier`, soit 27.

NB : cette instruction n'a modifié en rien la valeur de `ageOlivier` : une instruction d'affectation ne modifie que ce qui est situé **à gauche du signe =**.

- Affectation simple : type caractère

```
char sexe;
sexe = 'F';
```

Déclare une variable sexe de type **Char** puis lui affecte le caractère **F**.

Remarques : une valeur de type alphanumérique est toujours encadrée par des délimiteurs. Un caractère sera toujours délimités par des guillemets simple ('). Pourquoi ? Parce que la valeur 4 peut aussi bien représenter le nombre 4 que le caractère 4 noté '4'. Les délimiteurs permettent d'éviter toute ambiguïté à ce sujet.

2.2.Transtypage (type casting, ou cast en anglais).

Dans un langage fortement typé, l'utilisation de variables de types différents peut poser problème. Le transtypage permet de "mouler" une valeur d'un certain type dans un autre type donné.
En clair, le transtypage permet de convertir une donnée en un nouveau type : passer un **char** en **int**, un **byte** en **int**, un **float** en **int**...

Prenons quelques exemples :

```
char sexe;
sexe = (char)70;
```

Il ne s'agit pas d'une erreur, bien que cela puisse sembler curieux que l'on veuille transformer une valeur entière en une variable de type **char**. Dans ce cas, la valeur entière 70 se réfère au code décimal de la lettre **F** de la table Unicode. Par conséquent, cela revient à écrire : `sexe = 'F'`.

Soit le code problématique suivant :

```
int valeurInteger;
byte valeurByte;
valeurInteger = 129;
valeurByte = valeurInteger;
```

A la compilation, celui-ci produit l'erreur suivante :

`possible loss of precision`

Pourquoi cette erreur ? Tout simplement, le compilateur signale qu'il peut y avoir un problème lors du passage de la valeur d'un **int** vers un **byte**. En effet, un **byte** ne peut pas contenir tous les **int** car la taille de stockage d'un **byte** est plus petite (1 octet contre 4 pour un **int** !). La plage de valeur des **int** est plus grande que celle d'un **byte**. Par conséquent, c'est au programmeur d'assumer l'éventualité de ce problème, en autorisant explicitement la conversion comme suit :

```
int valeurInteger;
byte valeurByte;
valeurInteger = 129;
valeurByte = (byte)valeurInteger;
```

Maintenant, la compilation ne génère plus d'erreur et le programme pourra être exécuté. Remarquez quand même qu'à l'exécution un problème surviendra, car la valeur 129 qui est stockée sans problème dans `valeurInteger`, ne pourra pas être transférée dans `valeurByte`, car 129 est une valeur hors de la plage d'un Byte [-128 ; +127]. Ce n'est

plus le problème du compilateur, mais bien du programmeur, puisque qu'il a autorisé la conversion explicite.

Dans d'autres cas, la conversion est toujours implicite, et pour cause, elle n'est pas problèmatique. Par exemple, si mettre un **int** dans un **byte** nécessite une conversion explicite, l'inverse, mettre un **byte** dans un **int** ne pose pas de problème. En effet, un **int** est bien "assez grand" pour contenir tous les **byte**, et la conversion est alors implicite.

Le tableau suivant résume les cas de conversion implicite et explicite :

From-To	**byte**	**short**	**int**	**long**	**float**	**double**	**char**
byte 1 octet		Implicite	Implicite	Implicite	Implicite	Implicite	(**char**)
short 2 octets	(**byte**) ⚠		Implicite	Implicite	Implicite	Implicite	(**char**)
int 4 octets	(**byte**) ⚠	(**short**)		Implicite	Implicite	Implicite	(**char**) ⚠
long 8 octets	(**byte**) ⚠	(**short**)	(**int**)		Implicite	Implicite	(**char**) ⚠
float 4 octets	(**byte**) ⚠ ≈	(**short**) ⚠ ≈	(**int**) ⚠ ≈	(**long**) ⚠ ≈		Implicite	(**char**) ⚠
double 8 octets	(**byte**) ⚠ ≈	(**short**) ⚠ ≈	(**int**) ⚠ ≈	(**long**) ⚠ ≈	**(float)** ⚠		(**char**) ⚠
char 2 octets	Implicite ⚠⚠	Implicite ⚠⚠	Implicite	Implicite	Implicite	Implicite	

⚠ : Éventuel débordement de la mémoire de stockage de la variable cible, donc perte d'information. Mais pas d'erreur à la compilation

⚠⚠ : Erreur à la compilation lors d'un débordement de la mémoire.

≈ : Éventuelle perte de précision (ex: perte des chiffres après la virgule, utilisation d'un réel calculé)

Remarque : La conversions d'un réél en entier se fait en arrondissant à l'entier inférieur le plus proche. Dans l'exemple suivant « entier » aura comme valeur **3**:

```
int entier;
double reel;
reel = 3.84;
entier = (int)reel;
```

Attention : les constantes littérales (cf. chapitre suivant) de nombre réel sont par défaut du type **double**, il faut utiliser le suffixe **f** pour le réel soit du type **float.** Exemple **:**

```
float reel;
reel = 3.84f;
```

2.3.Les constantes.

L'utilisation de variables comme espace mémoire dont le contenu peut changer au cours de l'exécution d'un programme est fondamental en programmation. Cependant, il peut être utile d'avoir un conteneur dont la valeur ne change pas, par abus de langage, une sorte de "variable invariante". On appelle ce type de conteneur une **constante**.
Par exemple, on veut écrire un programme d'expérimentation en chimie qui utilise la température Celsius du point d'ébullition. Comment garantir que cette valeur ne sera pas modifiée ? En utilisant une constante.

Pour déclarer (et initialiser) une constante, la syntaxe est la suivante (par convention les constantes s'écrivent en **majuscule** en utilisant des **_** pour séparer les mots):

```
static final type NOM_DE_LA_CONSTANTE = valeur;
```

Par exemple :

```
static final int POINT_EBULLITION = 100;
```

Remarques :

- Pour vous convaincre, essayons de réattribuer une valeur à une constante.

```
static final int POINT_EBULLITION = 100;
POINT_EBULLITION = 120;
```

A la compilation, l'erreur suivante est générée :

cannot assign a value to final variable POINT_EBULLITION

- Lorsque vous écrivez :

```
short unAge;
unAge = 27;
```

27 est une valeur exprimée sous la forme d'une **constante littérale**. En effet, on ne peut plus la modifier lors de l'exécution (on dit que la valeur est "codée en dur" ou "hardcodée").

2.4.Opérateurs.

Un opérateur est un symbole qui entraine une action. Une action peut être l'affectation d'une valeur à une variable, l'addition de deux valeurs, la comparaison de deux valeurs, la concaténation de deux chaînes de caractères, etc.

Dans les sections précédentes, nous avons déjà vu un opérateur fondamental, celui d'affectation, permettant d'attribuer une valeur à une variable :

```
int ageOlivier;
ageOlivier = 27;
```

Dans la suite, nous verrons les opérateurs classiques de la programmation. Ces opérateurs ont un équivalent dans tous langages (bien que leur syntaxe puisse varier).

2.4.1.Opérateurs numériques

Ce sont des opérations arithmétiques classiques :

- addition : **+**
- soustraction : **-**
- multiplication : *****
- division : **/**
- reste de division entière (modulo): **%**

Exemple :

```
int ageJulien;
int ageOlivier;
ageOlivier = 27;
ageJulien = ageOlivier + 3;
```

Après l'exécution de ces trois instructions, `ageJulien` vaut maintenant 30, soit le résultat de l'addition exprimée à droite du signe =. De même que précédemment, `ageOlivier` vaut toujours 27.

Les opérateurs +, -, *, /, % fonctionnent donc comme vous pouvez l'imaginer, la multiplication et la division ayant « naturellement » priorité sur l'addition et la soustraction.

De plus, les parenthèses peuvent être utilisées, avec les mêmes règles qu'en mathématiques. Les parenthèses ne sont ainsi utiles que pour modifier cette priorité naturelle. Cela signifie qu'en informatique, 12 * 3 + 5 et (12 * 3) + 5 valent strictement la même chose, à savoir 41. En revanche, 12 * (3 + 5) vaut 12 * 8 soit 96.

Division usuelle (ou division dans les réels) :

Java utilise le même opérateur / pour la division entière et la division dans les réels. Dans l'exemple suivant le résultat n'est donc pas le plus intuitif:

```
11 / 5
```

Le résultat de cette divison (entière) est 2 (et non 2.2). En effet, java considère / comme un opérateur de division entière car les deux opérandes sont entières.

L'opérateur de modulo (**%**) nous permet d'obtenir le reste de cette division entière, c'est à dire 1 dans notre exemple.

Pour forcer la division dans les réels, il faut quer l'une des deux opérandes soit de type réel (double ou float). Pour ce faire il suffit de faire un cast (transtypage) sur l'une des opérandes. Comme c'est un cas très fréquent, il existe un raccourcis d'écriture pour faire le transtypage d'une constante littérale:

```
11 / 5d
```

2.4.2.Opérateurs conditionnels

Il s'agit des opérateurs de la logique booléenne. Rappelons ceux-ci par leurs tables de vérité :

ET logique

C1 **&&** C2		C1	
		true	false
C2	true	true	false
	false	false	false

OU logique

C1 **||** C2		C1	
		true	false
C2	true	true	true
	false	true	false

NON logique

C1	true	false
!(C1)	false	true

Quelques rappels :

- une expression ET est vraie si et seulement si ses deux opérandes sont vraies. Elle sera donc fausse si l'un au moins est faux.
- une expression OU est vraie si l'un au moins de ses opérandes est vrai. Elle ne peut être fausse que si les deux opérandes sont fausses.

La syntaxe Java de ces opérateur est la suivante

ET logique: **&&**
OU logique: **||**
NON logique: **!**

2.4.3.Opérateurs de comparaison.

Ces opérateurs permettent de comparer les valeurs de deux variables. Le **résultat** d'une telle expression de comparaison est un booléen, **true** ou **false**.

<	inférieur strict
<=	inférieur ou égal
==	égalité

>=	supérieur ou égal
>	supérieur strict
!=	différent

L'exemple suivant illustre une utilisation de ces opérateurs :

```
//déclaration d'une variable booléenne
boolean estPlusGrand;

//déclaration et initialisation des variables de taille
//deux entiers en centimètres
int tailleOlivier = 178;
int tailleSkis = 184;

//affichage des valeurs courantes
System.out.println("Olivier mesure " + tailleOlivier + " cm");
System.out.println("Les skis mesurent " + tailleSkis + " cm");

//affichage de la question
System.out.println("Les skis sont-ils plus grands qu'Olivier ?");

//évaluation de la comparaison en fonction des valeurs courantes
estPlusGrand = tailleSkis > tailleOlivier;

//affichage de la réponse
System.out.println("Réponse : " + estPlusGrand);
```

Remarque : Ne confondez pas l'opérateur d'affectation **=** avec l'opérateur de comparaison **==**

2.4.4.Informatique vs. mathématiques.

Revenons sur la trompeuse similitude de vocabulaire entre les mathématiques et l'informatique. En mathématiques, une « variable » est généralement une inconnue. Lorsque j'écris que y = 3 x + 2, les « variables » x et y satisfaisant à l'équation existent en nombre infini (graphiquement, l'ensemble des solutions à cette équation dessine une droite). Lorsqu'on écrit $ax^2 + bx + c = 0$, la « variable » x désigne les solutions à cette équation, c'est-à-dire 0, une ou deux valeurs à la fois… En informatique, une variable a toujours une valeur et une seule. Cette valeur justement, ne « varie » pas à proprement parler (du moins ne varie-t-elle que lorsqu'elle est l'objet d'une instruction d'affectation).
Remarque importante : nous veillerons toujours à ce qu'une variable soit au moins affectée une fois (on dit aussi initialisée) avant sa première utilisation dans une expression.

Une deuxième remarque concerne le signe de l'affectation. En algorithmique, comme on l'a vu, c'est le signe =. La quasi totalité des langages emploie ce signe égal. La confusion est donc facile avec les mathématiques. En mathématiques, A = B et B = A sont deux propositions strictement équivalentes. En informatique, absolument pas, puisque cela signifie respectivement, A affecté de B et B afffecté de A, deux choses bien différentes.

2.5.Remarque importante.

Notez bien, il est tout à fait possible d'écrire ce qui suit :

```
short compteur;
compteur = 0;
compteur = compteur + 1;
```

On peut donc affecter à une variable le résultat d'une expression utilisant la valeur de cette même variable avant affectation. Dans l'exemple, `compteur = 0 + 1`, donne 1.

Ce qui peut être compris comme suit :

«Nouvelle valeur de compteur» = «valeur actuelle de compteur» + 1

Ne soyez donc pas étonné par cette écriture, c'est tout-à-fait juste, et même très souvent utilisé.

2.6.Séquence et ordre des instructions.

Un algorithme est donc une séquence, c'est-à-dire une suite d'instructions. Ces instructions s'effectuent les unes après les autres dans l'ordre indiqué par l'algorithme. On parle de traitement séquentiel.

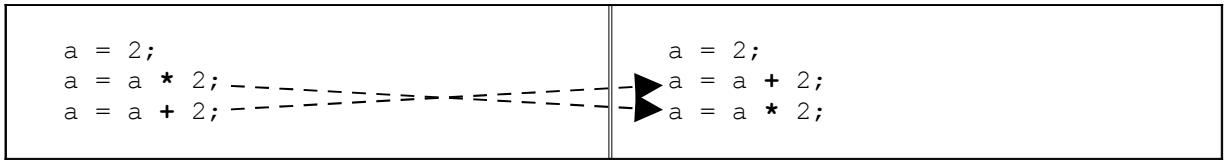
Rappelons le squelette de base de tout programme :

```
//Permet l'utilisation des fonctions de base du langage
import java.util.*;

public class NomDuProgramme {

    //Le bloc main est le point de démarrage de l'exécution du programme
    public static void main(String[] args) {
        //déclaration des variables. Par exemple:
        static int age;
        //d'autres déclaration peuvent suivre
        //instruction 1
        //instruction 2
        //...
        //instruction n
        //résultats
    }
}
```

Chacune de ces instructions peut être une instruction simple, ou une séquence, une instruction de sélection, une instruction de répétition. On ne commence pas l'exécution d'un algorithme au milieu du code, et on n'effectue pas de branchement (saut) incontrôlé. Un algorithme a un début, une fin, et seules les instructions de sélection et instructions de répétition sont autorisées à ordonner des branchements.

Nous l'avons déjà dit, l'ordre dans lequel les instructions sont écrites est important, il va jouer un rôle essentiel dans le résultat final.

Prenons l'exemple suivant :



Le résultat calculé n'est pas le même. Dans la séquence 1, **a** vaut 6 après calcule, alors que dans la séquence 2, en inversant simplement deux instructions, **a** vaudra 8 !
Il est tout aussi clair que ceci ne doit pas étonner : lorsqu'on indique le chemin à quelqu'un, dire « prenez tout droit sur 1km, puis à droite » n'envoie pas les gens au même endroit que si l'on dit « prenez à droite puis tout droit pendant 1 km ».

3. Opérations d'écriture et de lecture.

3.1.De quoi parle-t on ?

Imaginons que nous ayons fait un programme pour calculer le carré d'un nombre. Nous aurons par exemple l'algorithme suivant :

```
import java.util.*;

public class CalculCarre {

    public static void main(String[] args) {
       int carre;
       carre = 12 * 12;
    }
 }
```

Ce programme calcule en effet le carré de 12. Mais si l'on veut le carré d'un autre nombre que 12, il faut réécrire le programme. D'autre part, le résultat est calculé par la machine et stocké en mémoire, sans que l'utilisateur ne puisse voir le résultat.

Notre programme est une machine à calculer sourde et muette !

C'est pourquoi un certain nombre d'instructions permettent à la machine de dialoguer avec l'utilisateur du programme et inversement :

- **[Entrée]** : dans un sens, cela permet à l'utilisateur de saisir des valeurs au clavier pour qu'elles soient utilisées par le programme. C'est l'opération de lecture.

- **[Sortie]** : dans l'autre sens, cela permet au programme de communiquer des valeurs à l'utilisateur en les affichant à l'écran. C'est l'opération d'écriture.

Remarque : à première vue, on peut trouver cela illogique. Mais, notons que ces instructions doivent être considérées par rapport à la machine et non à l'utilisateur.

En Java nous utiliserons les composants **System** et **Scanner** pour gérer les entrées et sorties.

3.2.L'instruction d'écriture.

Le composant **System** met à disposition un ensemble de procédures d'affichage :

- **`System.out.print`** : pour afficher un contenu
- **`System.out.println`** : idem, mais avec un retour à la ligne après affichage

Utilisation : `System.out.print("Veuillez saisir une somme en CHF : ");`
On demande à **`System`** d'affichier à l'écran le message entre guillemets.

3.3.L'instruction de lecture.

Avant de pouvoir utiliser les fonctionnalités de **Scanner** il faut tout d'abord lui préciser que l'on veut lire les données au clavier. Ceci ce fait de la manière suivante:

```
Scanner read = new Scanner(System.in);
```

Dans le composant **Scanner** il existe une fonction de lecture par type de valeur à saisir. Chaque fonction récupère ce qui est saisi au clavier pour être ensuite transféré dans une variable. Il faut donc toujours prévoir une variable de stockage des données saisies.

- `read.nextInt()`: pour saisir un **int**
- `read.nextDouble()`: pour saisir un **double**
- ...

3.4.Un exemple.

```
import java.util.*;

public class CalculDeChange {
    //constantes
    static final double TAUX_CHANGE = 0.59;

    public static void main(String[] args) {
       //lecture
       Scanner read = new Scanner(System.in);
       System.out.println("Veuillez saisir une somme en CHF : ");
       double sommeCHF = read.nextDouble();
       //calcul de change
       double sommeEuro = sommeCHF * TAUX_CHANGE;
       //affichage du résultat
       System.out.println("Valeur en euro: " + sommeEuro);
    }
 }
```

Notez l'utilisation de l'opérateur de concaténation. C'est un opérateur qui va « coller » ensemble le contenu à gauche et le contenu à droite du **+**.

Tout programme interactif digne de ce nom devra toujours considérer les trois parties que sont,

- **l'entrée des données (saisie),**
- **les traitements de ces données**
- **la sortie des résultats (affichage).**