

Initiation à un langage informatique :

JAVA

De nombreux exemples et remarques de ce cours sont tirés de celui de Didier Ferment :
<http://www.u-picardie.fr/~ferment/initiation>

1. Du problème à son exécution

Le cycle de développement d'un "programme (ou d'une application) informatique " peut se résumer ainsi :
Problème → Analyse → Algorithme → Programme → Compilation → Exécution

Problème :

Par exemple, tracer la courbe représentative d'une fonction sur un intervalle donné.

Analyse :

Phase de réflexion qui permet d'identifier les caractéristiques du problème à traiter puis de découper le problème en une succession de tâches simples et distinctes.

Algorithme :

Description des opérations élémentaires à mettre en oeuvre (langage algorithmique) expliquant comment obtenir un résultat à partir de données.

Programme :

Fichier texte des instructions et de leurs enchaînements dans un langage informatique (ex: JAVA) que peuvent interpréter des ordinateurs et qui permettent de résoudre le problème posé.
Un programme utilise un ou plusieurs algorithmes.

Compilation :

Transformation en langage machine (suivant la plate-forme : Windows, Mac, etc...) d'un programme écrit en langage évolué : JAVA, C, COBOL, BASIC, PASCAL, FORTRAN,

Exécution :

L'ordinateur exécute les instructions d'un programme en langage "binaire".
Ainsi, l'utilisateur "lance" l'exécution de programme compilé.

2. Un premier algorithme

Un algorithme est une résolution en un certain nombre d'étapes "simples" d'un problème défini.

Problème : résolution de l'équation $ax^2 + bx + c = 0$.

Un algorithme de résolution de cette équation pourrait commencer par l'étude de son ordre et se décrire ainsi :

1. Si a est égal à 0,
on doit résoudre l'équation $b x + c = 0$.
2. Si a est égal à 0 et b est égal à 0,
la solution est n'importe quel réel si $c = 0$.
Il n'y a pas de solution sinon.
3. Si a est égal à 0 et si b n'est pas égal à 0,
la solution est $x = - c / b$.
4. Si a n'est pas égal à 0,
on commence par calculer le discriminant.
5. Si le discriminant est nul,
la solution est $- b / (2 * a)$
6. Si le discriminant est positif,
il y a 2 solutions réelles
7. Si le discriminant est négatif,
il y a 2 solutions complexes

3. Langage algorithmique

Un langage informatique permet la description de la résolution d'un problème en utilisant des opérations et des enchaînements d'opérations qui sont ceux des ordinateurs sans toutefois être particulier à un ordinateur précis.

Les actions de base d'un ordinateur :

- Mémoriser de l'information : un nombre entier, un nombre réel, une chaîne de caractères (une phrase), ..., plusieurs nombres, ... grâce à sa mémoire vive, ses disques etc.
- Déposer et lire cette information dans la mémoire grâce aux bus de la carte-mère.
- Réaliser des opérations sur les valeurs (informations) stockées en mémoire : addition de 2 entiers, multiplication d'un entier par un réel, ajouter (concaténer) un mot à une phrase, ... grâce à l'unité arithmétique et logique ALU du processeur.
- Comparer des valeurs en mémoire : telle valeur est-elle égale à telle autre?, inférieure? grâce au processeur.
- Enchaîner des actions selon divers modes : séquentiel (une action puis l'autre), alternatif (après une comparaison, 1 seule sur 2 actions possibles est réalisée), répétitive (une action est répétée autant de fois que nécessaire); ceci grâce aux jeux d'instructions du processeur.
- Communiquer avec l'extérieur, en particulier avec un être humain, via des périphériques : afficher une valeur sur l'écran, saisir une phrase au clavier,
- Coder (encoder et décoder) l'information : l'homme manipule des données numériques réelles en base dix ou des phrases de lettres alors que l'ordinateur ne manipule que des paquets (mots) de binaires (0 ou 1) .

Un programme exécutable est une séquence d'instructions pour le processeur.

Ces actions de base communes à tous les ordinateurs constituent les fondements du langage informatique que nous allons utiliser.

4. Variables

La base de l'informatique est le stockage et la manipulation des données. Ceci se fait essentiellement par l'utilisation de la variable (stockage) et l'instruction d'affectation (transfert d'information).

Une variable est la donnée de trois composantes :

- un identificateur
- un type
- une valeur

Une image simple d'une variable est simplement une boîte de rangement. Le nom correspond à l'étiquette sur la boîte, le type à la taille (ou forme) de la boîte et enfin la valeur à ce qu'il y a dans la boîte.

Les variables stockent (mémo-risent) les données traitées par les instructions. Ces données pourront être modifiées lors de l'exécution du programme.

4.1 Identificateur

Une variable est repérée par son identificateur (nom).

- En JAVA, l'identificateur (nom) d'une variable est composé de lettres, de chiffres, du caractère \$ et du caractère _ (blanc souligné ou underscore) mais il ne peut pas commencer par un chiffre.

a, x, X, temp, tEmp, soluceEquation, soluce_equation, soluce1, s1_4_Xt, unReel sont des identificateurs valides.

soluc ; %e n'est pas un identificateur valide.

- L'espace est un séparateur. On ne peut donc pas appeler une variable lundi matin.
- Il faut éviter les mots-clés déjà utilisés dans les instructions du langage.

abstract	boolean	break	byte	byvalue	case
cast	catch	char	class	const	continue
default	do	double	else	extends	false
final	finally	float	for	future	generic
goto	if	implements	import	inner	instanceof
int	interface	ong	native	new	null
operator	outer	package	private	protected	public
rest	return	short	static	sctrictfp	super
switch	synchronized	this	throw	throws	transient
true	try	var	void	volatile	while

- JAVA est sensible à la casse, c'est à dire qu'il différencie les minuscules des majuscules. Par exemple, Une_variable et une_variable sont des noms différents.
- En pratique, pour nommer des variables, la communauté JAVA utilise la règle suivante :
 - la première lettre est une minuscule.
 - chaque mot successif dans le nom de la variable commence par une majuscule.
 - toutes les autres lettres sont des minuscules.Par exemple, lundiMatin.
- Quand un identificateur est formé de plusieurs mots, on peut aussi utiliser le caractère _ . Par exemple, solutionEquation peut aussi s'écrire solution_equation.

4.2 Types

En langage JAVA, il faut préciser la nature des données qu'une variable va stocker. Toute variable possède donc un type (et une représentation), ainsi elle occupe un nombre d'octets connus (et dépendant du type de données stockées).

En JAVA, toutes les variables doivent être déclarées mais être aussi initialisées avant d'être utilisées. Si vous utilisez une variable sans l'avoir initialisée, le compilateur génère une erreur.

Il existe huit types simples de variables.

Nous utiliserons, en plus, deux autres types de variables : les chaînes de caractères et les tableaux. Les tableaux seront étudiés dans un prochain paragraphe.

Les types entiers :

Il existe plusieurs représentations des nombres entiers en fonction des valeurs qu'ils prennent. Les entiers sont signés par défaut, cela signifie qu'ils comportent un signe.

byte	Entier très court Codage sur 1 octet = 8 bits soit $2^8 = 256$ valeurs Plage de valeurs : de -128 à 127
short	Entier court Codage sur 2 octets = 16 bits soit $2^{16} = 65\,536$ valeurs Plage de valeurs : de -32 768 à 32 767
int	Entier Codage sur 4 octets = 32 bits soit $2^{32} = 4\,294\,967\,296$ valeurs Plage de valeurs : de -2 147 483 648 à 2 147 483 647
long	Entier long Codage sur 8 octets = 64 bits soit $2^{64} = 18\,446\,744\,073\,709\,551\,616$ valeurs Plage de valeurs : de -9 223 372 036 854 775 808 à 9 223 372 036 854 775 807

Les types réels :

En fait, les nombres que nous allons appeler réels sont des nombres à virgule flottante. Un nombre à virgule flottante est un nombre dans lequel la position de la virgule est repérée par une partie de ses bits (appelée l'exposant), le reste des bits permettent de coder le nombre sans virgule (la mantisse).

Il existe deux types de nombres à virgule flottante :

float	Plage de valeurs : de 1.40239846E-45 à 3.40282347E+38 Les nombres de type float sont codés sur 4 octets = 32 bits répartis de la façon suivante : 23 bits pour la mantisse 8 bits pour l'exposant 1 bit pour le signe
double	Plage de valeurs : de 4.9406545841246544E-324 à 1.79769313486231570E+308 Les nombres de type double sont codés sur 8 octets = 64 bits répartis de la façon suivante : 52 bits pour la mantisse 11 bits pour l'exposant 1 bit pour le signe

Le type caractère :

char	Tout caractère Unicode (lettres, nombres, ponctuations et autres symboles). Le codage Unicode utilise 16 bits (soit 2 octets) pour représenter un caractère et permet donc, théoriquement, de coder 65536 caractères en particulier les caractères accentués et tout autre caractère provenant d'autres alphabets (Cyrillique, Hébreux, Arabe, Grec, ...).
------	---

Le type chaîne de caractères :

String	Toute suite d'éléments de type char
--------	-------------------------------------

Les chaînes de caractères ne correspondent pas à un type de données mais à une classe, ce qui signifie qu'une chaîne de caractères est un objet possédant des attributs et des méthodes.

Le type booléen :

boolean Deux valeurs possibles : `true` (vrai) ou `false` (faux).
Codage sur 1 bit

4.3 Valeur et affectation

Lorsqu'une variable est déclarée, on peut lui affecter une valeur à l'aide de l'opérateur d'affectation "=".

La syntaxe de cette instruction est la suivante : `Nom_de_la_variable = donnee;`
L'affectation "=" de variable modifie la valeur d'une variable (à gauche) selon l'expression (à droite).

Pour stocker le caractère B dans la variable que l'on a appelée `deuxiemeCarAlpha`, il faudra écrire :

```
deuxiemeCarAlpha = 'B';
```

Ce qui signifie stocker la valeur ASCII de "B" dans la variable nommée "`deuxiemeCarAlpha`".
Il est bien évident que la variable doit être de type `char`.

L'affectation efface le précédent contenu d'une variable. Par exemple, lorsque l'on effectue la suite d'instructions suivante,

```
x = 3;  
x = 5;
```

La variable `x` (qui doit (pas exactement) être de type entier) prend successivement les valeurs 3 et 5.

Dans les exemples précédents, les valeurs 'B', 3, 5 sont appelés des littéraux.
Un littéral est un nombre, un caractère, un texte ou toute information représentant directement une valeur.
Plus simplement, cela signifie que la valeur saisie ou évaluée est celle qui est stockée.

On distingue cinq grands types de constantes littérales avec différentes représentations.

Les littéraux entiers :

Base décimale (base 10) :

C'est l'utilisation usuelle des entiers. L'entier est représenté par une suite de chiffres unitaires (de 0 à 9) ne devant pas commencer par le chiffre 0.

Par exemple : 210.

Base octale (base 8) :

L'entier est représenté par une suite d'unités incluant uniquement des chiffres de 0 à 7 et devant commencer par 0.

Par exemple : 014 (qui correspond à 12 en base 10).

Base hexadécimale (base 16) :

L'entier est représenté par une suite d'unités qui comprennent 16 caractères : les chiffres de 0 à 9 et les lettres de A à F (ou a à f). Un entier en base hexadécimale doit commencer soit par 0x soit par 0X.

Par exemple: 0xf1a3, 0xF1a3F, 0XF1A3f (qui correspond à 989759 en base 10).

On peut imposer une constante littérale entière d'être du type long en la suffixant par la lettre L (ou l).

Exemple : 1234L

Les littéraux réels :

Un littéral réel i.e. nombre à virgule flottante peut être :

- un entier décimal.

Par exemple : 895

- un nombre comportant un point (et non pas une virgule) ce que l'on appelle un représentation décimale.

Par exemple : 845.32 ou 895.0

- un nombre exponentiel, c'est-à-dire un nombre (éventuellement à virgule i.e. un point) suivi de la lettre e (ou E), puis d'un entier correspondant à la puissance de 10 (signé ou non, c'est-à-dire précédé d'un + ou d'un -) ce que l'on appelle un représentation scientifique. Par exemple : 2.75e-2, 35.8E+10 ou .25e-2

On peut imposer la constante réelle d'être du type float en la suffixant avec un F (ou f).

Les littéraux caractères :

Les littéraux de type caractère sont représentés entourés d'apostrophes.

La représentation d'un caractère peut être :

- soit normale : un seul caractère. Par exemple : 'A'.

- soit Unicode : 4 chiffres hexadécimal maximum précédés du sigle \u. Par exemple : '\u00FF'.

Il existe des codes caractères spéciaux :

Code	Signification
\'	'
\"	"
\t	tabulation
\n	nouvelle ligne
\b	retour arrière
\r	retour chariot

Les littéraux chaînes de caractères :

Ils sont constitués de tous les caractères de type char inclus au sein de guillemets.

Par exemple : "Le langage Java\n"

Les littéraux booléens :

On utilise les deux seules valeurs `true` et `false` (sans guillemets).

5. Instructions et opérateurs

5.1 Les instructions et la trace d'une exécution

Une instruction est une information qui permet de lancer une action.

Dans la précédente partie, nous avons vu l'instruction d'affectation.

Syntaxiquement parlant, les instructions se terminent par un point-virgule (pour séparer les choses) et ce même si l'instruction est en dernière position d'une construction (le rôle du ; change selon les langages informatiques : il est conseillé de les mettre car c'est obligatoire en JAVA).

Les instructions sont exécutées séquentiellement : l'une après l'autre (et dans l'ordre!). Du fait de cet ordre séquentiel, des séquences d'instructions dont on a interverti l'ordre peuvent avoir des effets très différents.

Par exemple, supposons que x et y sont des variables de type entier et étudions les deux séquences :

Séquence 1	Séquence 2
$x = 4;$	$x = 4;$
$y = 5;$	$y = 5;$
$x = y;$	$y = x;$
$y = x;$	$x = y;$

Lorsque l'on exécute la suite d'instructions de la séquence 1, à la fin, les variables x et y contiennent toutes les deux la valeur 5 alors que dans la séquence 2, les variables x et y contiennent toutes les deux la valeur 4.

Ici les séquences sont courtes et composées d'instructions simples. Nous aurons à étudier des séquences d'instructions plus complexes. Pour cela, on crée un historique, souvent sous forme de tableau, de la trace temporelle d'une exécution : les instructions exécutées, les valeurs de variables modifiées,

La numérotation des lignes du "programme" simplifie la lecture du tableau.

Chaque ligne du tableau est une photo instantanée de l'exécution : le résultat d'un test ou une nouvelle valeur pour une variable s'il y a lieu.

L'expression est une opération fondamentale en JAVA. Une expression est une instruction entraînant la production d'une valeur. Elle est construite avec des opérandes et un opérateur et elle retourne un résultat. Un opérande est une donnée sur laquelle l'opérateur s'applique. Il peut être une constante littérale, une valeur retournée par une fonction, ou une autre expression. Les opérateurs sont des symboles qui permettent de manipuler des variables, c'est-à-dire effectuer des opérations, les évaluer, ...

Une expression permet de calculer des valeurs. En composant les instructions, on définit ainsi le comportement que le programme devra adopter. Ce type d'instruction constitue le composant essentiel de base du transfert de données dans le monde informatique.

On distingue plusieurs types d'opérateurs:

5.2 Les opérateurs arithmétiques

Ce sont les opérateurs de calcul classiques : $+$, $-$, $*$, $/$ et le reste de la division euclidienne : $\%$.

Cependant, il faut bien noter que le même sigle (par exemple $+$) correspond à des opérateurs qui s'appliquent sur des types différents. Par exemple, l'addition est définie pour chacun des types entiers mais aussi pour la classe `String`. Mais lorsque l'on additionne deux entiers courts, le résultat est affecté à un entier court. Si ce sont deux réels de type `double`, on obtient un réel qui est aussi de type `double`. Si ce sont deux chaînes de caractères, on concatène celles-ci pour en former une troisième.

L'opération $4 + 3$ prend deux entiers et rend l'entier 7 avec la même représentation (même codage).

L'opération $3.2 + 1.84$ prend deux opérandes de type réel et renvoie le flottant 5.04 du même type.

L'opération `"Hello" + " world"` renvoie la chaîne `"Hello world"`.

De façon générale, il y a autant d'opérateurs $+$ que de types numériques.

De même, la division $/$ est une division réelle si les deux opérandes sont réelles. C'est une division entière si les deux opérandes sont entiers.

Par exemple, $14.0 / 4.0$ renvoie 3.5 alors que $14 / 4$ renvoie 3 car $14 = 4 \times 3 + 2$. Cette dernière égalité permet de signaler que $14 \% 4$ renvoie 2 .

Notons tout de suite qu'une expression peut être une instruction (si elle est suivie d'un point-virgule). Il est donc tout à fait possible d'avoir comme instruction $a + b$; malheureusement le résultat se "perd" car il n'est affecté à aucune variable.

5.3 Les opérateurs d'incrémentatation

Ce sont les opérateurs $++$ et $--$.

Ce type d'opérateur permet de facilement augmenter ou diminuer d'une unité une variable. Ces opérateurs servent souvent dans les boucles qui nécessitent un compteur qui augmente de un en un.

Si une variable x possède la valeur 7 , l'instruction $x++$; a pour effet de passer la valeur de x à 8 . De façon similaire, l'instruction $x--$; a pour effet de passer la valeur de x à 6 .

5.4 Les opérateurs de comparaison

De même qu'il existe des expressions qui calculent des résultats numériques, il en existe qui calculent des résultats booléens (c'est à dire un résultat pouvant être interprété soit comme vrai, soit comme faux). Il existe donc des opérateurs booléens nécessaires à la construction de ces expressions.

Nous y trouvons notamment des prédicats (autre façon de nommer ces opérateurs) de comparaison.

Les opérateurs de comparaison $==$ (égal) $!=$ (différent) $<$ (strictement inférieur) $>$ (strictement supérieur) $<=$ (inférieur ou égal) $>=$ (supérieur ou égal) comparent 2 opérandes de même type numériques (entier avec entier, réel avec réel, booléen avec booléen (les chaînes de caractères?)) (des conversions peuvent exister) et renvoient une valeur booléenne `true` ou `false`.

Attention à ne pas confondre l'opérateur de comparaison $==$ avec l'opérateur d'affectation $=$. Remarquons aussi que l'expression $a < b < c$; est erroné car elle fait intervenir 3 opérandes et 2 opérateurs.

5.5 Les opérateurs logiques (booléens)

Ce type d'opérateur permet de vérifier la véracité de conditions complexes (i.e. non simples).

Nous utiliserons essentiellement l'opérateur de négation $!$ qui est un opérateur unaire (qui ne prend qu'un seul opérande booléen), le $||$ (ou) et le $&&$ (et) qui prennent deux opérandes booléennes.

Les tableaux de vérité de ces opérateurs sont les suivants :

P	!P
true	false
false	true

A	B	A B
true	true	true
true	false	true
false	true	true
false	false	false

A	B	A && B
true	true	true
true	false	false
false	true	false
false	false	false

5.6 Opérateurs et affectation

Comme nous l'avons déjà vu, les opérateurs effectuent une opération. Il arrive souvent que le résultat de cette opération ait besoin d'être affecté à une variable.

La syntaxe est la suivante :

```
Nom_de_la_variable = expression a evaluer;
```

L'expression est évaluée : les variables sont remplacées par leurs valeurs, les opérations sont effectuées et la variable reçoit le résultat de l'expression calculée.

Par exemple, dans l'instruction $x = b / a$; la valeur de b est divisée par la valeur de a et la valeur obtenue est "affectée" à la variable x .

Voyons un exemple d'utilisation d'affectation avec des booléens :

```
estEgal = (a == b);  
eval = ((a < c) || (a > b));
```

Dans ces instructions, on suppose que `estEgal` et `eval` sont des variables booléennes et que a , b et c sont des variables entières par exemple.

Un exemple d'utilisation avec des chaînes de caractères :

```
s = "bon";  
t = s + "jour";  
t = t + " le monde";
```

Dans ces instructions, on suppose que s et t sont des chaînes de caractères. A la fin de l'exécution de ces trois instructions, la variable s contient la valeur "bon" et la variable t contient la valeur "bonjour le monde".

Imaginons que nous devons calculer la valeur du polynôme $5x^3 - x^2 + 1$ quand $x = 2$. Pour cela, nous disposons de trois variables x , xn et p de type double. Nous supposons, de plus, que la variable x a été précédemment affectée de la valeur 2.

Il reste à vérifier que la suite d'instructions suivantes satisfait bien au problème.

```
01    p = 1;  
02    xn = x * x;  
03    p = p - xn;  
04    xn = xn * x;  
05    p = p + 5 * xn;
```

Pour cela nous allons retracer l'exécution dans un tableau.

numéro de ligne	variables		donnée
	p	xn	x
			2
01	1		
02		4	
03	-3		
04		8	
05	37		

A la fin de l'exécution, la valeur de `p` est 37 c'est-à-dire le résultat désiré.

La précision des nombres réels est approchée. Elle dépend du type de réels (`float` ou `double`). Les résultats des opérations numériques sont donc des calculs approchés.

Avec les variables de type entier, il faut faire attention au débordement. Par exemple, si `i` est de type `byte` (de -128 à 127) et qu'il est affecté à 127 alors après l'exécution de l'instruction `i = i + 2;` la valeur de variable `i` est -127!

5.7 Les opérateurs d'assignation

Ces opérateurs permettent de simplifier des opérations telles que ajouter une valeur dans une variable et stocker le résultat dans la variable. Par exemple, avec l'opérateur d'assignation `+=`, il est possible d'écrire l'instruction `x = x + 2;` sous la forme `x += 2;`.

Les autres opérateurs du même type sont les suivants : `--` `*=` `/=`.

Le tableau suivant donne les équivalences entre ces opérateurs d'assignation et l'opérateur d'affectation classique :

<code>a += b;</code>	<code>a = a + b;</code>
<code>a -= b;</code>	<code>a = a - b;</code>
<code>a *= b;</code>	<code>a = a * b;</code>
<code>a /= b;</code>	<code>a = a / b;</code>
<code>a %= b;</code>	<code>a = a % b;</code>

Rappelons qu'un opérateur de type `x++` permet de remplacer les notations (plus lourdes) que sont `x = x + 1` et `x += 1`.

5.8 Les priorités

Lorsque l'on associe plusieurs opérateurs, il faut que le compilateur sache dans quel ordre les traiter, voici donc dans l'ordre décroissant les priorités des opérateurs que nous utiliserons :

1. `() []` .
2. `++ -- ! -` (avec un seul opérande : moins unaire)
3. `* / %`
4. `+ -`
5. `< <= > >=`
6. `== !=`
7. `&&`
8. `||`
9. `= += -= *= /= %= &=`
10. `,`

Si des opérateurs ont même priorité, on évalue l'expression de gauche à droite.

Etant donné la complexité de la règle précédente, il est préférable d'ajouter quelques parenthèses '(' et ')' pour être plus sûr des expressions utilisés (d'autant plus que les règles ne sont pas nécessairement les mêmes d'un langage à un autre).

Mais attention tout de même : l'utilisation abusive des parenthèses rend très rapidement un code illisible. Dans le but de ne pas arriver dans de telles situations, il faut quand-même mieux connaître quelques unes de ces règles. Ainsi, les expressions `3 * 2 + 4`, `(3 * 2) + 4` et `3 * (2 + 4)` ont respectivement les valeurs 10, 10 et 18. Les parenthèses dans la deuxième expression sont inutiles.

Il existe des cas où les parenthèses évitent les erreurs d'interprétation.

Par exemple les expressions : $x / j * 3;$
 $i * - j + 3 * i \% j + 4;$
sont équivalentes respectivement à : $(x / j) * 3;$
 $(i * (-j)) + ((3 * i) \% j) + 4;$

5.9 Les autres opérateurs du langage

Il existe d'autres opérateurs que nous serons amenés à utiliser.

Les opérateurs d'entrées-sorties :

`System.out.println()` : affiche à l'écran le littéral ou la valeur de la variable de type chaîne de caractères à l'intérieur des deux parenthèses.

`Clavier.lireInt()` : saisie les chiffres tapés au clavier jusqu'au retour chariot.

`Clavier.lireDouble()` : idem précédemment mais pour lire les réels.

Les fonctions mathématiques :

Elles sont situées dans la classe `Math`. En voici quelques exemples :

syntaxe	mathématiques	nombre d'opérande(s)	type d'opérande(s)	type du résultat
<code>Math.PI</code>	π	0		double
<code>Math.E</code>	e	0		double
<code>random()</code>		0		double
<code>abs()</code>	valeur absolue	1	int	int
<code>min(,)</code>	minimum	2	int	int
<code>max(,)</code>	maximum	2	int	int
<code>abs()</code>	valeur absolue	1	double	double
<code>sin()</code>	sinus	1	double	double
<code>cos()</code>	cosinus	1	double	double
<code>exp()</code>	exponentiel	1	double	double
<code>log()</code>	logarithme	1	double	double
<code>sqrt()</code>	racine carrée	1	double	double

Des opérateurs pour les tableaux :

`new` : opérateur d'allocation mémoire

`[]` : accès aux éléments d'un tableau

`length` : opérateur de taille

6. Affectation et type

6.1 Déclaration des variables

En JAVA, pour pouvoir utiliser une variable, il faut la définir, c'est-à-dire lui donner un nom et un type afin qu'un espace mémoire conforme au type de donnée qu'elle contient lui soit réservé.

Une variable doit être déclarée une seule fois.

La syntaxe pour déclarer une seule variable est :

```
type Nom_de_la_variable;
```

Par exemple : `int i;`

La syntaxe pour déclarer plusieurs variables du même type est :

```
type Nom_de_la_variable1, Nom_de_la_variable2, ...;
```

Par exemple : `int i, j;`

Java permet de définir une variable à n'importe quel endroit du programme, afin de le rendre plus lisible. Toutefois, une convention veut que la déclaration des variables se fasse plutôt en début de bloc.

Étudions un exemple simple de déclaration de variables et d'utilisation de celles-ci dans une petite séquence d'instructions :

```
double temp, x;
String ch;
x = 3.4;
temp = x - 0.6;
ch = "hello" + " world";
```

Si on reprend ces instructions pas à pas, on a :

- La déclaration de deux variables de type réel ayant pour identificateurs `x` et `temp`.
- La déclaration d'une variable de type chaîne de caractères d'identificateur `ch`.
- L'affectation de la valeur réelle `3.4` à la variable `x`.
- L'utilisation de la valeur de la variable `x` dans l'expression `(x - 0.6)`.
- Le résultat de l'expression calculée est `2.8`.
- L'affectation de la valeur `2.8` à la variable `temp`.
- Le calcul de l'expression `"hello" + " world"` c'est-à-dire la concaténation des deux chaînes, le résultat est la chaîne `"hello world"`.
- Pour terminer, l'affectation de `"hello world"` à la variable `ch`.

Le problème de la déclaration des variables semble simple pourtant il peut générer de nombreuses erreurs. Voici un exemple d'une suite d'instructions de déclaration :

```
int i, j;
double a, b, x;
int k;
String s, t;
boolean b;
z = 6;
i = t;
String t;
```

Nous nous trouvons en présence de 4 erreurs (classiques) :

- La variable `z` n'est pas déclarée
- On affecte une valeur chaîne (`t`) à une variable entière (`i`).
- La variable `t` est déclarée deux fois.
- La variable `b` est déclarée deux fois : une fois comme réel et l'autre comme booléen.

6.2 Initialisation des variables

La déclaration d'une variable ne fait que "réserver" un emplacement mémoire où stocker la variable. Dans de nombreux langage, tant que l'on n'a pas affecté de valeur à cette variable, celle-ci contient ce qui se trouvait précédemment à cet emplacement, que l'on appelle le garbage (détritus en français).

Par exemple, dans la suite d'instructions :

```
double x, temp;  
temp = x + 3;
```

La valeur de `temp` n'est pas déterminée puisque la variable `x` n'est jamais affecté !

Il ne faut donc jamais oublier de donner une valeur initiale à une variable : on parle d'initialisation.

On peut affecter une valeur initiale à la variable lors de sa déclaration. La syntaxe est la suivante :

```
type Nom_de_la_variable = valeur;
```

Par exemple, la suite d'instructions

```
float toto = 125.36f;  
String s = "Hello World";  
int i = 0;
```

est équivalente à

```
float toto;  
String s;  
int i;  
toto = 125.36f;  
s = "Hello World";  
i = 0;
```

6.3 Portée des variables

Selon l'endroit où l'on déclare une variable, celle-ci pourra être accessible (visible) partout dans le code ou bien que dans une portion restreinte de celui-ci, on parle de portée (ou visibilité) d'une variable.

Une variable déclarée à l'intérieur d'un bloc d'instructions (dans une fonction, entre des accolades ou une boucle par exemple) aura sa portée limitée à l'intérieur du bloc (et de ses sous-blocs) dans lequel elle est déclarée c'est-à-dire qu'elle est inutilisable ailleurs, on parle alors de variable locale.

6.4 Définition des constantes

Une constante est une variable dont la valeur est invariante lors de l'exécution d'un programme. En JAVA, le mot clé `final` permet de définir une variable dont la valeur ne peut pas être modifiée après son initialisation.

Par exemple, l'instruction, `final int test = 1;` aura pour effet de définir une variable `test` de type entier possédant la valeur 1 et ne pouvant pas être modifiée dans la suite du programme, sinon le compilateur générerait une erreur.

6.5 Les conversions de types

Dans cette partie, nous allons préciser les règles sur le typage.

Comme nous l'avons vu, une variable doit être affectée par une valeur de son type, sinon une erreur peut se produire. De même, une expression possède un type. Le type de l'expression est déterminé par les types des opérandes de l'expression. Rappelons, par exemple, que les opérateurs arithmétiques `+`, `-`, `*`, `/` renvoie une valeur qui est du même type que les opérandes utilisés. Lorsque l'on additionne deux entiers, on obtient un entier. Mais la question est alors de savoir ce qui se passe si l'on tente d'ajouter un entier avec un décimal. Il semble naturel que le résultat doit aussi être décimal.

JAVA est un langage fortement typé. En général, il n'est pas permis de convertir un type en un autre. Mais, il y a une conversion implicite d'un type numérique de petite taille vers un type de plus grand taille. La modification du type de donnée est effectuée automatiquement par le compilateur.

Par exemple, les opérations `+` `-` `*` `/` appliquées sur un opérande entier et un opérande réel convertissent l'entier en réel puis effectue l'opération réelle.

De plus, l'opération `+` appliquée sur un opérande chaîne et un opérande numérique convertit le numérique en chaîne puis effectue l'opération concaténation.

Etudions, par exemple, la suite d'instructions :

```
double x;  
int a;  
String result;  
a = 3;  
x = 4.0;  
x = a * x;  
result = "la valeur de x est " + x;
```

1. L'opération `*` entre l'entier `a` et le réel `x` s'effectue ainsi :
 - conversion de la valeur entière 3 de `a` en une valeur réelle 3.0.
 - multiplication réelle des deux réels 3.0 et 4.0, pour donner un résultat réel.
2. L'opération `+` entre la chaîne "la valeur de x est " et la valeur de `x` est la concaténation :
 - le réel 12.0 est convertit en chaîne de caractères "12.0"
 - la concaténation est effectuée pour donner "la valeur de x est 12".

Voici encore un exemple de conversion implicite :

```
int e;  
double r;  
String ch;  
e = 2;  
r = e;  
ch = "r =" + r;
```

On peut résumer en disant que la conversion d'une valeur d'un certain type vers un type plus "vaste" (entier vers réel, réel vers chaîne, entier vers chaîne, ...) est une conversion qui ne fait pas perdre d'information : elle est implicite (automatique). Par exemple, une valeur de type `byte` qui est comprise entre -128 et 127, peut "tenir" dans une variable de type `short` (de -32 768 à 32 767).

Les conversions peuvent se faire au sein d'une instruction et il faut bien vérifier le type des résultats des expressions.

```

int i,j,k,p;
double x,y;
i = 5;
j = 2;
x = 5.0;
k = i / j;
y = x / j;
p = i % j;

```

La variable `k` prend la valeur 2 car `/` est l'opérateur de la division entière puisque `/` agit sur des entiers. Dans le même ordre d'idée, la variable `x` prend la valeur 2.5 car `/` est la division réelle. La variable `p` prend la valeur 1 car `%` est l'opérateur reste (modulo) de la division entière à résultat entier.

Dans le cas d'une conversion d'un type grand vers un type faible, si le risque de perdre des chiffres significatifs est grand, Java refusera la conversion. Il faudra mettre en place une conversion explicite.

Pour convertir (avec possibilité de perte) un réel en type entier, on utilise la fonction `(int)` dont l'exemple qui suit donne la syntaxe à utiliser : il suffit de préfixer la valeur par le type, mit entre parenthèses.

Soit la suite d'instructions :

```

int e;
double r;
r = 2.5;
e = (int)(0.6 + 2 * r);

```

L'expression `0.6 + 2 * r` est évaluée en une valeur réelle 5.6, puis une opération de conversion en entier est effectuée pour obtenir l'entier 5. Enfin l'entier 5 est affectée à la variable entière `e`. C'est une conversion avec perte d'information puisqu'à partir de l'entier 5, on ne peut pas savoir si le réel correspondant était 5.6 ou 5.8 ou

Il n'y a pas de conversion possible du type `boolean` en un autre type non chaîne de caractères.

7. Les instructions conditionnelles et les blocs d'instructions

Une condition est une expression qui renvoie une valeur booléenne .

On appelle structures conditionnelles les instructions qui permettent de tester si une condition est vraie ou fausse et qui possèdent un comportement qui prend en compte la véracité de la condition. Ces structures conditionnelles peuvent être associées à des structures répétitives suivant la réalisation de la condition, on appelle ces structures des boucles.

7.1 L'instruction if

L'instruction `if` permet d'exécuter une série d'instructions lorsqu'une condition est réalisée.

La syntaxe de cette expression est la suivante:

```

if (condition)
    instruction_a_executer_quand_la_condition_est_vrai;

```

Attention, puisque majuscules et minuscules sont différentes en JAVA, `IF` ou `If` ou bien encore `iF` sont des identificateurs différents.

condition est une expression à valeur booléenne c'est-à-dire dont la valeur sera true ou false. L'instruction `instruction_a_executer_quand_la_condition_est_vrai;` sera exécutée uniquement si la valeur du test est true.

La condition doit être entre parenthèses et il faut faire attention à ne pas mettre de point-virgule après. Il est possible de définir plusieurs conditions à remplir avec les opérateurs `&&` et `||`.

Par exemple, l'instruction

```
if ((condition1)&&(condition2))
    instruction_a_executer_si_vrai;
```

n'exécutera l'instruction `instruction_a_executer_si_vrai;` que si les conditions `(condition1)` et `(condition2)` sont toutes les deux vraies.

Dans le même ordre d'idée, l'instruction

```
if ((condition1)|| (condition2))
    instruction_a_executer_si_vrai;
```

n'exécutera l'instruction `instruction_a_executer_si_vrai;` que si l'une des deux conditions `(condition1)` ou `(condition2)` est vraie.

L'instruction `if` dans sa forme basique ne permet de tester qu'une condition, or la plupart du temps on aimerait pouvoir choisir les instructions à exécuter en cas de non réalisation de la condition. Il est donc possible de rajouter le mot clé `else` suivi d'une instruction qui sera alors lancée en cas de non-réalisation de la condition.

La syntaxe est la suivante:

```
if (condition)
    instruction_a_executer_quand_la_condition_est_vrai;
else
    instruction_a_executer_quand_la_condition_est_fausse;
```

Soient les séquences suivantes sachant que `a`, `x` et `y` sont des entiers.

	séquence 1		séquence 2
01	<code>if (x <= 0)</code>		<code>if (x <= 0)</code>
02	<code> a = 2;</code>		<code> a = 2;</code>
03	<code> a = 3;</code>		<code>else</code>
04	<code> y = a + 2;</code>		<code> a = 3;</code>
05			<code> y = a + 2;</code>

Retraçons la trace de l'exécution de ce deux séquences une première fois avec la variable `x` affectée à `-1` et la deuxième fois affectée à `1` :

numéro de ligne	Séquence 1				Séquence 2			
	variables			test	variables			test
	a	x	y		a	x	y	
		-1						
01				true			true	
02	2				2			
03	3							
04			5					
05						4		

numéro de ligne	Séquence 1				Séquence 2			
	variables			test	variables			test
	a	x	y		a	x	y	
		1						
01				false				false
02								
03	3							
04			5		3			
05						5		

7.2 Les blocs d'instructions

Il peut-être nécessaire de regrouper plusieurs instructions à un endroit précis. Or certaines constructions (instructions), n'accepte qu'une seule sous-instruction (comme, par exemple, pour le `if`). La solution à ce problème a été de définir un bloc d'instructions comme étant qu'une seule instruction.

Au point de vue de la syntaxe, un bloc d'instructions commence par une accolade ouvrante '{' et se termine par une accolade fermante '}'. Entre ces deux caractères, nous y trouvons des instructions classiques, qui sont séparées les unes des autres par des points-virgules.

La syntaxe d'un bloc d'instructions est :

```
{
    instruction_1;
    instruction_2;
    ...
    instruction_n;
}
```

L'exécution du bloc d'instructions consiste en l'exécution séquentielle (l'une après l'autre, dans l'ordre) des n instructions. L'instruction `instruction_i;` peut être une instruction simple, un bloc (dans ce cas elle n'est pas suivie du `;`) ou encore une instruction `if` contenant 1 ou 2 blocs d'instructions. Un bloc d'instructions peut apparaître partout où une instruction simple apparaît.

Attention : la dernière instruction simple d'un bloc doit se terminer, elle aussi, par un point-virgule. Le bloc peut contenir zéro (bloc vide), une (équivalent à une instruction unique) ou plusieurs instructions.

7.3 L'instruction if et les blocs d'instructions

Comme nous l'avons signalé précédemment toutes les instructions simples de l'instruction `if` peuvent être remplacées par des blocs d'instructions.

Exemple d'une séquence d'instructions comportant des blocs :

```
if (i == 0)
{
    j = 3;
    k = 5;
}
else
{
    j = 2;
    k = 4;
}
m = j + k;
```

Selon que la valeur de i soit nulle ou non, m prendra la valeur 8 ou 6.

Certains problèmes nécessitent l'imbrication d'instructions `if`. Par exemple, considérons le problème de résoudre une équation $ax + b = 0$ (sachant que a et b seront des données réelles pour le programme).

L'étude des différentes valeurs de a et b amène à la solution suivante :

```
String soluce; double x;
if (a == 0)
    if (b == 0)
        soluce = "n'importe quel réel convient";
    else
        soluce = "aucune solution";
else
    {
        x = -b / a;
        soluce = "x =" + x;
    }
System.out.println(soluce);
```

Exécution de cette séquence pour l'équation $2x - 5.4 = 0$.

Dans ce cas, a vaut 2 et b vaut -5.4 .

La condition (`a == 0`) est fausse.

La variable `x` reçoit la valeur 2.7 et la variable `soluce` reçoit la chaîne "`x = 2.7`".

Exécution de cette séquence pour l'équation $0x + 2 = 0$.

Dans ce cas, a vaut 0 et b vaut 2.

La condition (`a == 0`) est vraie et la condition (`b == 0`) est fausse.

La variable `soluce` reçoit "`aucune solution`".

Exécution de cette séquence pour l'équation $0x + 0 = 0$.

Dans ce cas, a vaut 0 et b vaut 0.

La condition (`a == 0`) est vraie et la condition (`b == 0`) est vraie.

La variable `soluce` reçoit "`n'importe quel réel convient`".

7.4 L'instruction `switch`

Il est fréquent en informatique d'avoir à comparer la valeur d'une variable à un certain nombre de valeurs. Cela pourrait se faire par l'imbrication d'un certain nombre d'instructions `if`. En JAVA, on peut regrouper ces actions en une seule instruction : `switch`. Cette instruction permet de faire plusieurs tests de valeurs sur le contenu d'une même variable. Cela simplifie beaucoup la lisibilité du code.

L'instruction `switch` se comporte comme un aiguillage mais ne fonctionne que sur des types énumérables simples (`byte`, `short`, `int`, `long` ou `char`)!

La syntaxe est la suivante :

```
switch (variable)
{
    case Valeur_1 :
        Liste d'instructions
        break;
    case Valeur_2 :
        Liste d'instructions
        break;
    ...
    case Valeurs_i :
        Liste d'instructions
        break;
    default :
        Liste d'instructions
        break;
}
```

Les parenthèses qui suivent le mot clé `switch` indiquent une expression dont la valeur est testée successivement par chacun des `case`. Lorsque l'expression testée est égale à une des valeurs suivant un `case`, la liste d'instruction qui suit celui-ci est exécutée.

Le mot clé `break` indique la sortie de la structure conditionnelle. Si ce mot n'est pas mis dans un `case`, toutes les instructions du `case` suivant (et éventuellement des autres) sont effectuées ceci jusqu'à ce que l'on rencontre soit un `break` soit l'accolade fermante `}` de la structure `switch`.

Le mot clé `default` est facultative. Elle précède la liste d'instructions qui sera exécutée si l'expression n'est égale à aucune des valeurs.

Il faut faire attention à ne pas oublier d'insérer des instructions `break` entre chaque test. Cet oubli est difficile à détecter car aucune erreur n'est signalée par le compilateur.

7.5 L'indentation, les commentaires et autres règles

Les programmes sont lus par les ordinateurs mais aussi par des hommes. Si l'ordinateur a son système de lecture (assez rigide) qu'il faut respecter, il est important de rendre le programme lisible pour un être humain. Aussi une présentation soignée aide-t-elle fortement à leur lecture.

L'indentation du texte (décalage des instructions par rapport au bord gauche de la page) est une convention facilitant la lecture.

Regardons une séquence d'instructions bâclée :

```
double x;
if (a == 0)
if (b == 0)
x = 0;
else
x = a;
else
x = b;
System.out.println(x);
```

La technique de l'indentation (décalage) facilite la lecture et la compréhension.

En voici, les règles :

- Les `if` et `else` de la même instruction sont alignés verticalement.
- Les instructions à l'intérieur d'une instruction `if` sont décalées vers la droite par rapport au `if`.
- Les accolades `{` et `}` d'un même bloc d'instructions sont alignées verticalement
- Les instructions à l'intérieur d'un bloc d'instructions sont en décalées vers la droite par rapport aux accolades.

Voici l'exemple précédent correctement indenté :

```
double x;
if (a == 0)
    if (b == 0)
        x = 0;
    else
        x = b;
else
    x = a;
System.out.println(x);
```

Remarquons que pour un ordinateur il n'y aurait pas de différence si la séquence d'instructions était écrite sous la forme suivante :

```
double x; if (a == 0) if (b == 0)x = 0; else x = b; else x = a;
System.out.println(x);
```

Les tabulations, les passages à la ligne ou les espaces supplémentaires (2 ou plus au lieu de 1) ne sont pas pris en compte par le compilateur.

L'imbrication de plusieurs instructions `if` crée souvent des ambiguïtés. C'est surtout le cas si ces instructions sont mal indentées comme dans l'exemple suivant :

```
x = 1;
if (a == 0)
    if (b == 0)
        x = 2;
else
    x = 3;
```

Si `a` est différent de 0, à la fin de la suite d'instructions, `x` vaut 1.

Si `a` est égal à 0 et `b` est différent de 0, à la fin de la suite d'instructions, `x` vaut 3.

En effet, la règle veut que le `else` se rapporte toujours au dernier `if` rencontré dans le bloc d'instruction.

La bonne indentation pour l'exemple ci-dessus était donc :

```
x = 1;
if (a == 0)
    if (b == 0)
        x = 2;
    else
        x = 3;
```

Dans l'exemple, si l'on avait voulu que le `else` s'adresse au premier `if`, on aurait été obligé d'ajouter des accolades de bloc.

```
x = 1;
if (a == 0)
    {
        if (b == 0)
            x = 2;
    }
else
    x = 3;
```

Un autre moyen pour améliorer la lisibilité d'un programme consiste à mettre des commentaires. Un commentaire est une information ajoutée à un programme mais qui ne fait pas partie des instructions exécutées par l'ordinateur.

Le langage JAVA possède trois styles de commentaires :

`// commentaire` Tout ce qui est entre `//` et la fin de ligne est ignoré .

`/* commentaire */` Tout le texte qui est entre `/*` et `*/` est ignoré.
Cette forme peut s'étendre sur plusieurs lignes.

`/** commentaire */` Cette forme de commentaire (identique à la forme précédente) sera traitée par le programme pour générer un fichier HTML de documentation du programme.

7.6 L'instruction while

Nous résolvons beaucoup de problèmes quotidiens avec des actions répétitives. Par exemple, enfoncer un clou dans un mur se réalise ainsi : taper avec un marteau sur la tête (du clou!) tant que le clou dépasse du mur. Il est clair que généralement on ne connaît pas le nombre de fois où il faudra taper sur le clou.

Nous avons ce type de problème en informatique.

Par exemple, considérons le problème mathématique qui consiste à calculer le ppcm de 2 entiers. Le ppcm (plus petit commun multiple) de 9 et 15 est 45.

Pour ce faire, un algorithme (c'est loin d'être le meilleur) consiste à essayer progressivement les multiples de 9, donc 9, 18, 27, 36, 45, 54 jusqu'à ce que l'un d'entre eux soit aussi multiple de 15 c'est-à-dire que le reste de la division euclidienne par 15 soit nul.

Il faut donc recommencer les actions (multiple suivant) tant que la condition de divisibilité n'est pas satisfaite : c'est ce que l'on appelle une répétition avec une condition arrêt.

Il existe plusieurs instructions qui nous permettent de mettre en place des boucles (ou des itérations, c'est la même chose) et d'en contrôler leurs exécutions. Trois styles de boucles sont utilisables, et deux instructions permettent d'interrompre l'exécution à l'intérieur d'une boucle.

L'instruction (la boucle) `while` est l'une de ces instructions répétitives. Sa syntaxe est la suivante :

```
while (condition)
    a_faire
```

Dans cette instruction, `(condition)` est une expression booléenne : c'est le test de la boucle appelé encore test de continuité ou test d'arrêt.

`a_faire` peut-être une instruction simple (et est alors suivie d'un point-virgule) mais est plus généralement un bloc d'instructions appelé corps de la boucle.

Le fonctionnement de la boucle est le suivant. En premier lieu, il y a évaluation de `(condition)`. Si sa valeur est `true`, alors on exécute `a_faire`. Puis on évalue à nouveau `(condition)`, si sa valeur est toujours `true`, on exécute une nouvelle fois `a_faire`. On continue ainsi de suite jusqu'à ce que `(condition)` prenne la valeur `false`.

```
int a,b;
a = 2;
b = 5;
while (a < b)
{
    b = b - a;
}
```

Pour exemple, considérons le problème mathématique qui consiste à calculer le ppcm de 2 entiers. Nous supposons que ces deux entiers sont les valeurs des variables `a` et `b` et que, pour la trace d'exécution, on leur a affecté les valeurs respectives 15 et 9 (plus subtile que 9 et 15).

```
01  int m;
02  m = a;
03  while (m % b != 0)
04      {
05          m = m + a;
06      }
06  System.out.println("le ppcm de " + a + " et " + b + " est " + m);
```

La trace de l'exécution de cette séquence d'instructions donne :

n°	test	données		variable
		a	b	m
		15	9	
2				15
3	true			
4				30
3	true			
4				45
3	false			
6	Le ppcm de 15 et 9 est 45			

Dans une instruction `while`, il faut faire attention à ne pas mettre de point-virgule après la condition.

Les instructions `while` sont imbriquables entre elles, avec des instructions `if` ou encore des blocs.

Il faut faire attention au critère d'arrêt car les risques de boucle infinie (boucle dont la condition reste toujours vraie) sont grands. Une erreur classique est l'absence dans le corps de la boucle d'une instruction qui fait évoluer le ou les paramètres du critère d'arrêt de la condition.

7.7 La boucle `do while`

Cette structure est très proche de la structure `while`.

Il est parfois utile d'effectuer au moins une fois un bloc d'instruction avant d'effectuer le test de la boucle. Même si on pourrait dupliquer le code de la boucle, l'instruction "`do while`" donne une réponse plus appropriée à ce problème.

Sa syntaxe est :

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

Dans cette boucle "`do while`", la condition est évaluée après l'exécution du corps de la boucle. Elle est au minimum exécutée une fois même si la condition à tester est fausse au départ.

8. Les boucles avec compteurs

8.1 La variable "compteur de boucle" et l'instruction `while`

Une boucle "avec compteur" est une structure qui permet d'exécuter un nombre prédéterminé de fois la même série d'instructions. Une boucle "avec compteur" nécessite une variable de contrôle de boucle ou de contrôle du nombre d'itérations. Une telle boucle peut être mise en place avec l'instruction `while`. Dans ce cas, la syntaxe est la suivante :

```

int compt,valeurDInitialisation,valeurDePas,valeurLimite;
...
compt = valeurDInitialisation;
while (comparaison de compteur avec valeurLimite)
{
    aFaire;
    compt = compt + valeurDePas;
}

```

La variable compteur `compt` est initialisée à sa valeur initiale avant la boucle.
 Une instruction dans la boucle incrémente ou décrémente cette variable à chaque itération de la boucle.
 La condition d'arrêt porte sur ce compteur par comparaison à une valeur limite, le corps de la boucle peut exploiter la valeur de ce compteur.

Dans la boucle avec compteur, le nombre d'itération de la boucle est prévisible.

```

int i = 100, j = 0, somme = 0;
while (j <= i)
{
    somme += j;
    j++;
}

```

A la sortie de la boucle de l'exemple précédent, la variable `somme` contient la somme des entiers de 0 à 100.

Par exemple, on cherche à calculer la somme des entiers compris entre 2 valeurs données affectées aux variables `inf` et `sup`, la séquence d'instructions suivante répond à ce problème :

```

01  int compt, som;
02  som = 0;
03  compt = inf;
04  while (compt <= sup)
05  {    som = som + compt;
06      compt = compt + 1;
07  }
08  System.out.println(som);

```

La trace d'exécution de cette séquence avec 74 et 76 affectés respectivement à `inf` et `sup` donne :

n°	test	paramètres		variables	
		inf	sup	compt	som
		74	76		
2					0
3				74	
4	true				
5					74
6				75	
4	true				
5					149
6				76	
4	true				
5					225
6				77	
4	false				
8	Affichage de 225				

Dans cet exemple, la variable compteur `compt` est initialisée à `inf`, le pas est 1 et la valeur limite est `sup`.

Remarquons que l'exécution `somme(73,56)` donne logiquement 0.

8.2 La boucle for

En JAVA, il existe une instruction spécialisée pour effectuer des boucles avec compteur. Il s'agit de l'instruction `for`.

La structure de la boucle `for` est la suivante :

```
for (exp1 ; exp2 ; exp3)
    a_faire
```

- `a_faire` est le corps de la boucle. C'est une instruction simple (auquel cas elle se termine par un point-virgule) soit un bloc d'instructions.
- `exp1` initialise un compteur (après une déclaration éventuelle auquel cas la portée de la variable compteur est limitée à la boucle)
- `exp2` est une condition de test qui est évaluée à chaque passage. La boucle est exécutée tant que `exp2` est vraie.
- `exp3` est exécutée après l'exécution du contenu de la boucle. Cette expression est la modification de compteur, souvent une incrémentation.

Par exemple :

```
for (int i = 1; i < 4; i++)
{
    System.out.println("i = " + i);
}
```

Cette boucle affiche :

```
    i = 1
    i = 2
    i = 3
```

Elle commence à `i = 1`, vérifie que `i` est bien inférieur à 4, etc... jusqu'à atteindre la valeur `i = 4`, pour laquelle la condition n'est plus vérifiée, la boucle s'interrompt.

Remarquons que JAVA autorise la déclaration de la variable de boucle dans l'instruction `for` elle-même.

Une boucle `for` de l'exemple précédent est alors équivalente à la structure `while` suivante :

```
int i = 1;
while (i < 4)
{
    System.out.println("i = " + i);
    i++;
}
```

Comme nous venons de le voir, on peut déclarer la variable de boucle dans l'instruction `for`. Puisqu'il est possible d'imbriquer toutes les structures itératives, il faut veiller à la portée des variables locales.

Il faut toujours vérifier que la boucle a une condition de sortie de correcte. En phase de test, une instruction `System.out.println()` dans la boucle est un bon moyen pour vérifier la valeur du compteur pas à pas en l'affichant.

Il faut aussi bien compter le nombre de fois que l'on veut faire exécuter la boucle:

```
for(i=0;i<10;i++)    exécute 10 fois la boucle (i de 0 à 9)
for(i=0;i<=10;i++)   exécute 11 fois la boucle (i de 0 à 10)
for(i=1;i<10;i++)    exécute 9 fois la boucle (i de 1 à 9)
for(i=1;i<=10;i++)   exécute 10 fois la boucle (i de 1 à 10)
```

8.3 Choix du type de boucle

Il existe une certaine méthodologie à adopter pour l'écriture d'une boucle. On peut commencer par faire soi-même le calcul sur un ou plusieurs exemples judicieusement choisis : cas généraux, cas particuliers.

Se pose ensuite le choix entre boucle avec compteur ou sans. La première question est donc : peut-on prévoir (déterminer) le nombre d'itérations? Si oui, une boucle avec compteur peut être utilisée sinon le choix se porte obligatoirement sur une boucle sans compteur. Dans ce dernier cas, la nouvelle question est faut-il commencer l'action avant de tester ou l'inverse ? S'il faut tester d'abord, on utilise une boucle `while`. Dans l'autre cas, ce sera une instruction `do while`.

Il ne faut pas oublier d'initialiser les variables utilisées (si elles sont nécessaires bien entendu), d'écrire les conditions d'arrêt, d'écrire l'incréméntation de la variable de contrôle s'il y en a.

Enfin, il faut mieux exécuter la boucle pour les cas extrêmes et au moins un cas "normal" voire même essayer de prouver que cela fonctionne dans tous les cas.

9. Structuration d'un programme

Nous venons de voir des structures répétitives. Il arrive aussi que nous ayons à faire de façon répétitive des opérations similaires mais avec des valeurs différentes. Bien sûr, on pourrait répéter le code, mais, dans JAVA, il existe une méthode plus agréable, il s'agit de l'utilisation de fonctions, plus précisément pour nous, les méthodes de classe (autrement appelé procédures ou sous-programmes dans d'autres langages).

Une fonction permet de regrouper des opérations concernant des objets qu'on lui fournit. Elle peut renvoyer un résultat ou simplement afficher un message à l'écran.

9.1 Les fonctions avec résultat

C'est le cas par exemple de la fonction racine carrée `sqrt` : on lui fournit un réel et elle renvoie un réel. Ce type de fonction calcule un résultat à partir de données appelées paramètres.

La syntaxe d'une fonction avec résultat est la suivante :

```
public static type_renvoi nom_fonct(liste_para)
{
    type_renvoi calc;
    a_faire
    return calc;
}
```

- Le terme `public` signifie que cette méthode est disponible pour différents programmes
- Le terme `static` permet de signaler qu'il s'agit d'une méthode de classe (les fonctions qui nous intéressent dans ce cours).
- La valeur renvoyé par la fonction sera de type `type_renvoi`. Elle ne pourra être affectée qu'à une variable de même type (ou alors cela nécessitera conversion).

- Les paramètres précédés de leur type et séparés par des virgules sont déclarés dans `liste_para`.
- `a_faire` est soit une instruction simple (auquel cas elle est suivie d'un point-virgule) ou un bloc d'instructions.
- La valeur renvoyée par la fonction est celle de la variable `calc` qu'il faudra déclarer dans le bloc d'instructions de la fonction. L'exécution de `return` met fin à l'exécution de la fonction.

La première partie qui décrit les caractéristiques d'utilisation de la fonction est appelée l'en-tête de la fonction. La seconde partie, celle entre les accolades est le corps de la méthode (fonction). Elle spécifie l'enchaînement des instructions nécessaires au calcul du résultat de la fonction.

Il est souvent utile de rajouter des commentaires à une fonction pour expliciter sa nature.

En général, les identificateurs (noms) des fonctions (comme celle des variables) commencent par une minuscule.

Etudions un exemple de fonction :

```
public static int nbrSolReel(double a,double b,double c)
/* renvoie le nombre de solutions réelles
de l'equation ax2+bx+c=0 avec la
pré-condition : a non nul */
{
    double d;
    int n;
    d = b * b - 4 * a * c;
    if (d == 0)
        n = 1;
    else
        if (d > 0)
            n = 2;
        else
            n = 0;
    return n;
}
```

La fonction `nbrSolReel` nécessite qu'on lui fournisse trois valeurs réelles. Celles-ci seront stockées dans les variables locales (à la fonction) `a`, `b` et `c`.

Elle renvoie une valeur de type entier qui pourra être affiché ou stocké (dans une variable adaptée).

De plus, nous supposons que nous avons vérifié que les valeurs fournies à la fonction vérifient bien la pré-condition.

Voyons quelques exemples d'utilisation (appel) de la fonction `nbrSolReel` avec l'équation $2x^2+x-1$:

```
int nbr;
nbr = nbrSolReel(2,1,-1);
System.out.println("Nbre de solution(s) : " + nbr);

System.out.println("Nbre de solution(s) : " + nbrSolReel(2,1,-1));

double aa,bb,cc;
int nbr;
aa = 2;
bb = 1;
cc = -1;
nbr = nbrSolReel(aa,bb,cc);
System.out.println("Nbre de solution(s) : " + nbr);
```

```

double a,b,c;
a = 2;
b = 1;
c = -1;
System.out.println("Nombre de solution(s) : " + nbrSolReel(a,b,c));

```

Tous les exemples précédents donnent le même résultat c'est-à-dire affichage à l'écran de :
 "Nombre de solution(s) : 2".

Énumérons (et/ou rappelons) quelques règles (simples) à respecter sur les paramètres et résultats de fonction tant du côté de la fonction appelée que de l'appelante :

- La liste des paramètres spécifie pour chaque donnée son type.
- Le type (des données ou du résultat) peut être réel, entier, chaîne de caractères, ou autre.
- Le nombre de paramètres de l'appel de la fonction doit être égal au nombre de paramètres de la déclaration de la fonction.
- Les types des paramètres de l'instruction appelante doivent être identiques avec ceux de la fonction.
- Lors de l'appel, ce sont les valeurs qui sont "passées" (fournies), donc les paramètres (données) de l'appel sont des expressions.
- La valeur est renvoyée par l'instruction `return` qui termine l'exécution de la fonction.
- Le type du résultat doit être égal au type de la valeur renvoyée.
- Dans l'expression de l'appelante, le type "évalué" doit être celui du type de résultat de la fonction.

Un autre exemple de fonction : nous cherchons à déterminer si une liste de 3 entiers est en ordre strict (croissant ou décroissant). La réponse est donc soit vrai ou faux i.e. `true` ou `false`.

```

01 public static boolean
02     enOrdreStrict(int a, int b, int c)
03     /* retourne vrai si a < b < c ou a > b > c */
04     {   boolean aInfb, cInfOuEgalb, result;
05         aInfb = (a < b);
06         cInfOuEgalb = (b >= c);
07         if (aInfb)
08             return !(cInfOuEgalb);
09         else
10             {   result = (b != a) && (b > c);
11                 return result;
12             }
13     }

```

Effectuons la trace d'exécution de l'appel `enOrdreStrict(7, 4, 4)`.

n°	test	paramètres			variables		
		a	b	c	aInfb	cInfOuEgalb	result
1-2		7	4	4			
5					false		
6						true	
7	false						
10							false
11	résultat renvoyé : false						

9.2 Les fonctions sans résultat

La syntaxe est la suivante :

```
public static void nom_de_la_fonction(liste_parametres)
{
    corps_de_la_méthode
}
```

Le terme `void` signifie que la fonction ne renvoie rien.

On peut utiliser un tel type de fonction pour des affichages de messages à l'écran par exemple. Supposons que nous avons utilisé la fonction `nbrSolReel` vue précédemment pour déterminer le nombre de solution réelle d'une équation $ax^2 + bx + c = 0$. On peut imaginer utiliser la fonction suivante pour signifier à l'utilisateur le résultat

```
public static void combien(int n)
/* affichage du nombre de solutions d'une équation de degré 2
   pre-requis n = 0 ,1 ou 2 */
{
    switch (n)
    {
        case 2 : System.out.println("Votre équation possède 2
                                solutions réelles");
                break;
        case 1 : System.out.println("Votre équation possède 1
                                solution réelle");
                break;
        case 0 : System.out.println("Votre équation ne possède
                                pas de solution réelle");
                break;
        default : System.out.println("Valeur de n incorrecte");
                 break;
    }
}
```

9.3 Une bibliothèque

La bibliothèque regroupe plusieurs fonctions de résolution de problèmes connexes qui seront facilement réutilisables. La notion de bibliothèque porte aussi d'autres noms : librairie, API, classe, package ...

Par exemple, dans JAVA, on trouve une bibliothèque `Math` dans laquelle sont regroupées les fonctions de résolutions mathématiques : racine carrée (`sqrt`), puissance (`power`), logarithme (`log`), cosinus (`cos`), etc.

En JAVA, la notion de bibliothèque est appelée `class`.

Sa syntaxe est relativement simple :

```
class Nom_de_la_biblio
{
    Une_ou_des_fonctions
}
```

Les noms (identificateurs) de `class` bibliothèque (ou programme) commence par une majuscule

Dans une classe bibliothèque, les fonctions seront obligatoirement spécifiées "`public static`", `public` pour qu'elles soient accessibles de partout, et `static` car elles servent dans le cadre d'une "bibliothèque boîte à outils" et non pas dans un objet de la `class`.

Une bibliothèque (class) est enregistrée (déclarée) dans un fichier texte dont l'identificateur est le plus souvent : `Nom_de_la_biblio.java` (même si un seul fichier texte peut contenir plusieurs classes et dans ce cas le nom du fichier ne pourra pas être égal à toutes les classes à la fois).

La classe devra être compilée pour pouvoir être utilisée : la commande `javac Nom_de_la_biblio` produira un fichier `Nom_de_la_biblio.class`.

Par contre cette classe n'est pas directement exécutable puisqu'elle ne contient pas de fonction nommée `main`.

Par exemple, on peut regrouper dans une bibliothèque `Equations` (que nous enregistrerons dans un fichier `Equations.java`) plusieurs fonctions de résolution d'équations. Ainsi est constituée une boîte à outils pour les équations.

```
class Equations
{
    public static int nbrSolReel(double a,double b,double c)
    /* renvoie le nombre de solutions réelles
    de l'equation ax2+bx+c=0 avec la
    précondition : a non nul */
    {
        instructions
    }

    public static void combien(int n)
    /* affichage du nombre de solutions d'une équation de degre 2
    pre-requis n = 0 ,1 ou 2 */
    {
        instructions
    }

    public static double
        calculSolution_AXPlusB (double a, double b)
    /* calcul de la solution de l'équation a x + b = 0
    Précondition : a != 0
    */
    {
        double x;
        x = - b / a;
        return x;
    }
}
```

9.4 Un programme ou une classe

La différence entre un programme et une bibliothèque est la présence obligatoire dans le premier d'une fonction `main` dont la syntaxe est la suivante :

```
public static void main(String[] args)
{
    corps_de_la_méthode
}
```

Cette fonction `main` sera celle qui sera "lancée" (exécutée) à l'appel du programme.

La fonction `main` prend un paramètre de type tableau de chaînes de caractères qui correspond aux arguments qui peuvent être associés à l'appel de la fonction.

Le terme `public` signifie que cette méthode est disponible pour d'autres classes.

De même que pour une bibliothèque, le nom du fichier texte contenant le programme est obligatoirement suivi de l'extension `.java`.

Il est préférable (pour retrouver facilement ses programmes) qu'il soit du même nom que celui de la classe du programme à la casse près (majuscule et minuscule).

Par convention, les identificateurs de programme (comme les bibliothèques) commencent par une lettre majuscule.

```
class Equation2ndDegre
/* Programme qui resoud l'equation ax2+bx+c=0 */
{
    public static void main (String args[])
    {
        double a, b, c, d, sol_1, sol_2;
        int nbr;
        System.out.println("Solutions réelles
                               de l'équation ax2+bx+c=0");
        System.out.println("donnez a non nul");
        a = Clavier.lireDouble();
        System.out.println("donnez b");
        b = Clavier.lireDouble();
        System.out.println("donnez c");
        c = Clavier.lireDouble();
        nbr = Equations.nbrSolReel(a,b,c);
        Equations.combien(nbr);
        d = delta(a,b,c);
        if (nbr == 2)
            {
                sol_1 = (-b - Math.sqrt(d)) / (2 * a);
                sol_2 = (-b + Math.sqrt(d)) / (2 * a);
                System.out.println("Les solutions sont " + sol_1
                                   + "et" + sol_2);
            }
        else
            if (nbr == 1)
                {
                    sol_1 = -b / (2 * a);
                    System.out.println("La solution est " + sol_1);
                }
            else
                System.out.println("Pas de solution réelle");
    }

    public static double delta(double a, double b, double c)
    {
        double d;
        d = b * b - 4 * a * c;
        return d;
    }
}
```

Notons tout de suite, deux types d'appels de fonction. Lorsque que la fonction dont on se sert, ne se trouve pas dans la même `class` (programme ou bibliothèque), il faut préciser la `class` où elle se trouve en préfixant le nom de celle-ci.

L'appel à la fonction `lireDouble` de la classe (bibliothèque) `Clavier` fonctionne en JAVA.

Pour simplifier le mécanisme de saisie des données, des enseignants de l'UPJV ont écrit puis compilé une classe "bibliothèque" contenant les fonctions de saisie `lireDouble`, `lireInt`, ...

9.5 Compilation et exécution

Soit le fichier texte `Progr1.java` de contenu :

```
class Progr1
{
    public static void main (String args[])
    {
        .....
    }
}
```

La compilation de ce programme source en un programme exécutable se fait par la commande `javac Progr1.java`.

Le compilateur analyse le programme : si des erreurs sont détectées alors elles sont affichées à l'écran. Si l'analyse se déroule bien alors le compilateur génère un programme exécutable dans un fichier nommé `Progr1.class`.

Même si le fichier s'appelait `toto.java`. La compilation `javac toto.java` aurait produit un fichier `Progr1.class` car la classe dans le fichier se nomme `Progr1`.

L'exécution du programme exécutable se fait par la commande `java Progr1`.

L'exécution des instructions du corps de fonction `main` du fichier `Progr1.class` est lancée.

10. Les tableaux

Il arrive régulièrement que les objets que nous avons à manipuler soient nombreux : le nom des mois de l'année, les jours de la semaine, une liste d'étudiants, leurs notes ...

Créer une variable pour chacun des objets peut alors être long et fastidieux et difficilement programmable pour un traitement général.

En JAVA, il existe une structure particulière permettant de mieux gérer ce genre de données : les tableaux. On peut déclarer et créer des tableaux de tout type.

La déclaration d'un tableau se fait suivant l'une des deux formes suivantes :

```
typeDuTableau nomDeTableau[]; ou typeDuTableau[] nomDeTableau;
```

`typeDuTableau` peut être n'importe quel type en particulier ce peut être aussi un tableau.

Par exemple : `String Etudiants[];`
`double[] notesEtudiants;`

Une fois le tableau créé, il faut l'initialiser c'est-à-dire préciser le nombre de valeurs que le tableau aura à stocker. C'est l'instanciation du tableau (de l'objet) : cela permet de réserver en mémoire (allocation) la place nécessaire aux éléments. Cela se fait par l'instruction :

```
nomDuTableau = new typeDuTableau [nombreDeValeurs]
```

Par exemple : `Etudiants = new String[20];`
`notesEtudiants = new double[20];`

Chaque élément dans le tableau est du type déclaré. Les éléments du tableau se comportent comme des variables avec, de plus, l'avantage de l'indice (c'est-à-dire leur numéro dans le tableau) qui permet des opérations "globales". Les éléments du tableau sont individuellement accessibles par leur indice grâce à l'instruction :

```
nomDuTableau[indice]
```

Les indices vont de 0 à `nombreDeValeurs - 1`.

Par exemple, la séquence d'instruction suivante créer un tableau de chaînes de caractères qui contient les jours de la semaine

```
String jourSemaine[];
jourSemaine = new String[7];
jourSemaine[0] = "lundi";
jourSemaine[1] = "mardi";
jourSemaine[2] = "mercredi";
jourSemaine[3] = "jeudi";
jourSemaine[4] = "vendredi";
jourSemaine[5] = "samedi";
jourSemaine[6] = "dimanche";
for (int i=1, i<=7,i++)
    System.out.println("Le "+i+"ème jour est "+jourSemaine[i-1]);
```

Java ne copie jamais implicitement des données complexes. Donc, un tableau passé en paramètre d'une fonction n'est jamais copié. C'est son adresse mémoire qui est passée en argument.

Le tableau possède un attribut (caractéristique, propriété) `nomDuTableau.length` il s'agit du nombre d'éléments déclaré.

Voyons maintenant un exemple complet de l'utilisation de fonction et boucle avec les tableaux.

```
class TestTableau
{
    public static void main (String args[])
    {
        int numMois, jourParMois[];
        jourParMois = initialiseMois(false);
        numMois = 0;
        while (numMois < jourParMois.length)
            {
                System.out.println(" le "+ (numMois+1) +"-ième mois de"
                    + "l'année comporte "+jourParMois[numMois]+" jours");
                numMois = nummois + 1;
            }
    }

    public static int[]
        initialiseMois(boolean anBissextile)
    /* initialise un tableau du nombre de jours pour
       chaque mois d'une année bissextile ou non */
    {
        int tabMois[];
        tabMois = new int[12];
        tabMois[0] = 31;
        tabMois[1] = 28;
        tabMois[2] = 31;
        tabMois[3] = 30;
        tabMois[4] = 31;
        tabMois[5] = 30;
        tabMois[6] = 31;
        tabMois[7] = 31;
        tabMois[8] = 30;
        tabMois[9] = 31;
        tabMois[10] = 30;
        tabMois[11] = 31;
        if (anBissextile) tabMois[1] = 29;
        return tabMois;
    }
}
```

11. Les variables "drapeaux"

Les variables "drapeaux" sont liées aux boucles à plusieurs conditions d'arrêt.

Jusqu'à maintenant les boucles que nous avons programmées sont à une seule condition d'arrêt. Avec les tableaux, nous allons avoir souvent besoin d'une condition d'arrêt liée au traitement à effectuer (cela ne sert à rien de continuer le traitement de tout le tableau lorsque l'on a obtenu le résultat voulu) et d'une condition d'arrêt liée à la boucle de traitement de tous les éléments de ce tableau.

Nous allons étudier un grand classique de l'algorithmique : rechercher la présence (l'occurrence) d'un élément dans un tableau.

Il va falloir (avec une boucle) :

- parcourir tous les éléments du tableau
- arrêter le parcours si on trouve cet élément (cela ne sert à rien de continuer)
- arrêter quand on a atteint le dernier élément du tableau (donc parcouru tout le tableau sans succès)

```
public static boolean
    occurrence(double tab[], double valeurCherchee)
/* retourne true si la valeur recherchée valeurCherchee
   est présente dans le Tableau tab et false sinon */
{
    int i;
    boolean fini, trouve;
    i = 0;
    trouve = false ;
    fini = false ;
    while (!fini)
        if (i >= tab.length)
            fini = true;
        else
            if (tab[i] == valeurCherchee)
                {
                    trouve = true;
                    fini = true;
                }
            else
                i = i+1;
    return trouve;
}
```