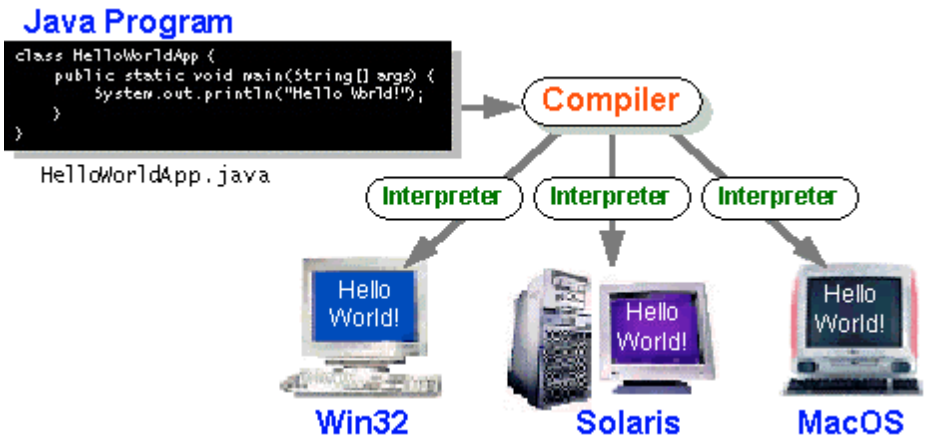
Java 2

Les fondements du langage Java



Le contenu de ce livre pdf de cours d'initiation à Java 2 est inclus dans un ouvrage papier de 1372 pages édité en Novembre 2004 par les éditions Berti à Alger.

http://www.berti-editions.com

L'ouvrage est accompagné d'un CD-ROM contenant les assistants du package pédagogique.

Rm di Scala

Corrections du 28.05.05

*Pour pouvoir s'initier à Java avec ce cours et à peu de frais dans un premier temps, il faut télécharger gratuitement la dernière version du **J2SE** (n° 1.5.0 depuis janvier) disponible sur le site de SUN à http://sun java.com, puis utiliser un environnement pédagogique permettant d'éditer, de compiler et d'exécuterdes programmes Java avec le J2SE ; le logiciel JGrasp de l'université d'Aburn est un tel environnement téléchargeable gratuitement (utiliser Google et taper Jgrasp pour connaître l'URL du site de téléchargement)*

Remerciements : (*pour les corrections d'erreurs*)

A Vecchio56@free.fr, internaute averti sur la syntaxe de base C++ commune à Java et à C#.

A mon épouse Dominique pour son soutien et sa patience qui me permettent de consacrer de nombreuses heures à la construction du package et des cours inclus et surtout comme la seule personne en dehors de moi qui a eu la constance de relire entièrement toutes les pages de l'ouvrage, alors que l'informatique n'est pas sa tasse de thé.

Remerciements : (*diffusion de la connaissance*)

- A l'université de Tours qui supporte et donne accès à la partie Internet du package pédagogique à partir de sa rubrique "cours en ligne", à partir duquel ce document a été élaboré.

- Au club des développeurs francophones qui héberge un site miroir du précédent et qui recommande le package pédagogique (http://rmdiscala.developpez.com/cours/) à ses visiteurs débutants.

Cette édition a été corrigée duran tles 2 mois l'été 2004, elle a été optimisée en nombre de pages papier imprimables.

Remerciements : (*anticipés*)

Aux lecteurs qui trouveront nécessairement des erreurs, des oublis, et autres imperfections et qui voudront bien les signaler à l'auteur afin d'améliorer le cours, **e-mail : discala@univ-tours.fr**

SOMMAIRE

Types, opérateurs, instructions

Structures de données de base

Java et la Programmation Objet

EXERCICES

Introduction aux bases de Java 2

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

Destiné au départ à la programmation de centraux téléphoniques sous l'appellation de langage "oak", la société Sun a eu l'idée de le recentrer sur les applications de l'internet et des réseaux. C'est un langage en évolution permanente Java 2 est la version stabilisée de java fondée sur la version initiale 1.2.2 du JDK (Java Development Kit de Sun : http://java.sun.com)

Les objectifs de java sont d'être multi-plateformes et d'assurer la sécurité aussi bien pendant le développement que pendant l'utilisation d'un programme java. Il est en passe de détrôner le langage C++ dont il hérite partiellement la syntaxe mais non ses défauts. Comme C++ et Delphi, java est algorithmique et orienté objet à ce titre il peut effectuer comme ses compagnons, toutes les tâches d'un tel langage (bureautiques, graphiques, multimédias, bases de données, environnement de développement, etc...). Son point fort qui le démarque des autres est sa portabilité due (en théorie) à ses bibliothèques de classes indépendantes de la plate-forme, ce qui est le point essentiel de la programmation sur internet ou plusieurs machines dissemblables sont interconnectées.

La réalisation multi-plateformes dépend en fait du système d'exploitation et de sa capacité à posséder des outils de compilation et d'interprétation de la machine virtuelle Java. Actuellement ceci est totalement réalisé d'une manière correcte sur les plates-formes Windows et Solaris, un peu moins bien sur les autres semble-t-il.

Notre document se posant en manuel d'initiation nous ne comparerons pas C++ et java afin de voir les points forts de java, sachons que dans java ont été éliminés tous les éléments qui permettaient dans C++ d'engendrer des erreurs dans le code (pointeurs, allocation-désallocation, héritage multiple,...). Ce qui met java devant C++ au rang de la maintenance et de la sécurité.

En Java l'on développe deux genres de programmes :

- Les ***applications*** qui sont des logiciels classiques s'exécutant directement sur une plate-forme spécifique soit à travers une machine virtuelle java soit directement en code exécutable par le système d'exploitation. (code natif).

- les ***applets*** qui sont des programmes java insérés dans un document HTML s'exécutant à travers la machine virtuelle java du navigateur lisant le document HTML.

Les types primitifs
Java 2

1 - Les types élémentaires et le transtypage

Tout n'est pas objet dans Java, par souci de simplicité et d'efficacité, Java est un langage fortement typé. Comme en Delphi, en Java vous devez déclarer un objet ou une variable avec son type avant de l'utiliser. Java dispose de même de types prédéfinis ou types élémentaires ou primitifs.

Les types servent à déterminer la nature du contenu d'une variable, du résultat d'une opération, d'un retour de résultat de fonction.

Tableau synthétique des types élémentaires de Java

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

Signalons qu'en java toute variable qui sert de conteneur à une valeur d'un type élémentaire précis doit préalablement avoir été déclarée sous ce type.

Il est possible d'indiquer au compilateur le type d'une valeur numérique en utilisant un suffixe :

- **l** ou **L** pour désigner un entier du type long
- **f** ou **F** pour désigner un réel du type float
- **d** ou **D** pour désigner un réel du type double.

Exemples :

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

Transtypage : opérateur ( )

Les conversions de type en Java sont identiques pour les types numériques aux conversions utilisées dans un langage fortement typé comme Delphi par exemple (**pas de conversion implicite**). Si vous voulez malgré tout, convertir une valeur immédiate ou une valeur contenue dans une variable il faut explicitement transtyper cette valeur à l'aide de l'opérateur de transtypage noté: **( )**.

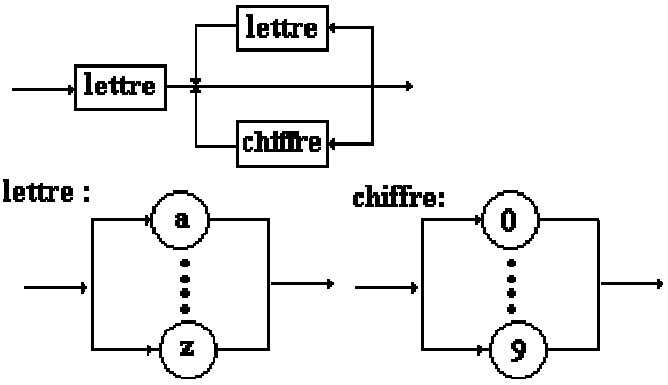
- **int** x ;
  x = **(int)** y ; signifie que vous demander de **transtyper** la valeur contenue dans la variable y en **un entier signé 32 bits** avant de la mettre dans la variable x.
- Tous les types élémentaires peuvent être transtypés à l'exception du type **boolean** qui ne peut pas être converti en un autre type (différence avec le C).
- Les conversions **peuvent être restrictives** quant au résultat; par exemple le transtypage du réel 5.27e-2 en entier (x = **(int)**5.27e-2) mettra l'entier zéro dans x.

2 - Variables, valeurs, constantes

Comme en Delphi, une variable Java peut contenir soit une valeur d'un type élémentaire, soit une référence à un objet. Les variables jouent le même rôle que dans les langages de programmation classiques impératifs, leur visibilité est étudiée ailleurs.

Les identificateurs de variables en Java se décrivent comme ceux de tous les langages de programmation :

Identificateur Java :



Attention Java est sensible à la casse et fait donc une différence entre majuscules et minuscules, c'est à dire que la variable **BonJour** n'est pas la même que la variable **bonjour** ou encore la variable **Bonjour**. En plus des lettres, les caractères suivants sont autorisés pour construire un identificateur Java : "$" , "_" , "µ" et les lettres accentuées.

Exemples de déclaration de variables :

int Bonjour ; **int** µEnumération_fin$;
float Valeur ;
char UnCar ;
boolean Test ;

etc ...

Exemples d'affectation de valeurs à ces variables :

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

Test = false ; **boolean** Test = false ;

Exemple avec transtypage :

```
int Valeur ;
char car = '8' ;
Valeur = (int)car - (int)'0';
```

fonctionnement de l'exemple :

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

Les constantes en Java ressemblent à celles du pascal

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

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

Une constante peut être déclarée et initialisée plus loin une seule fois :

final int x ; ….. x = 10;

Base de représentation des entiers

Java peut représenter les entiers dans 3 bases de numération différentes : décimale (base 10), octale (base 8), hexadécimale (base 16). La détermination de la base de représentation d'une valeur est d'ordre syntaxique grâce à un préfixe :

- **pas de préfixe ----> base = 10** ***décimal.***
- **préfixe 0 ----> base = 8** ***octal***
- **préfixe 0x ----> base = 16** ***hexadécimal***

3 - Priorité d'opérateurs

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

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

- Par exemple sur les entiers l'expression 2+3*4 vaut 14 car l'opérateur * est plus prioritaire que l'opérateur +, donc l'opérateur * est effectué en premier.
- Lorsqu'une expression contient des opérateurs de **même priorité alors Java effectue les évaluations de gauche à droite**. Par exemple l'expression 12/3*2 vaut 8 car Java effectue le parenthésage automatique de gauche à droite ((12/3)*2).

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

Les opérateurs
Java 2

1 - Opérateurs arithmétiques

opérateurs travaillant avec des opérandes à valeur immédiate ou variable

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

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

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

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

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

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

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

opérateurs travaillant avec une unique variable comme opérande

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

L'objectif de ces opérateurs ++ et --, est l'optimisation de la vitesse d'exécution du bytecode

dans la machine virtuelle Java.

post-incrémentation : k++

la valeur de k est d'abord utilisée telle quelle dans l'instruction, puis elle est augmentée de un à la fin. Etudiez bien les exemples ci-après ,ils vont vous permettre de bien comprendre le fonctionnement de cet opérateur.

Nous avons mis à côté de l'instruction Java les résultats des contenus des variables après exécution de l'instruction de déclaration et de la post incrémentation.

Exemple 1 :

int k = 5 , n ; **n = k++ ;**	**n = 5**	**k = 6**

Exemple 2 :

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

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

Exemple 3 :

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

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

Exemple 4 : Utilisation de l'opérateur de post-incrémentation en combinaison avec un autre opérateur unaire.
int nbr1, z , t , u , v ;

nbr1 = 10 ; **v = nbr1++**	**v = 10**	**nbr1 = 11**
nbr1 = 10 ; **z = ~ nbr1 ;**	**z = -11**	**nbr1 = 10**
nbr1 = 10 ; **t = ~ nbr1 ++ ;**	**t = -11**	**nbr1 = 11**
nbr1 = 10 ; **u = ~ (nbr1 ++) ;**	**u = -11**	**nbr1 = 11**

La notation " **(~ nbr1) ++** *" est refusée par Java.*
remarquons que les expressions **"~nbr1 ++ "** et **"~ (nbr1 ++)"** produisent les mêmes

effets, ce qui est logique puisque lorsque deux opérateurs (ici ~ et ++)ont la même priorité, l'évaluation a lieu de gauche à droite.

pré-incrémentation : ++k

la valeur de k est d'abord augmentée de un ensuite utilisée dans l'instruction.

Exemple1 :

int k = 5 , n ;

n = ++k ;	**n = 6**	**k = 6**

Exemple 2 :

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

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

Exemple 3 :

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

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

post-décrémentation : k--

la valeur de k est d'abord utilisée telle quelle dans l'instruction, puis elle est diminuée de un à la fin.

Exemple1 :

int k = 5 , n ;

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

pré-décrémentation : --k

la valeur de k est d'abord diminuée puis utilisée dans l'instruction.

Exemple1 :
int k = 5 , n ;

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

Reprenez avec l'opérateur - - des exemples semblables à ceux fournis pour l'opérateur ++ afin d'étudier le fonctionnement de cet opérateur (étudiez (- -k - k) et (k - - -k)).

2 - Opérateurs de comparaison

Ces opérateurs employés dans une expression renvoient un résultat de type booléen (**false** ou **true**). Nous en donnons la liste sans autre commentaire car ils sont strictement identiques à tous les opérateurs classiques de comparaison de n'importe quel langage algorithmique (C, pascal, etc...). Ce sont des opérateurs à deux opérandes.

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

3 - Opérateurs booléens

Ce sont les opérateurs classiques de l'algèbre de boole **{** { **V**, **F** }, ! , & , | **}** où { **V**, **F** } représente l'ensemble {**V**rai , **F**aux}.

Les connecteurs logiques ont pour syntaxe en Java : **! , & , | ,** ^ :

& : { **V**, **F** } x { **V**, **F** } → { **V**, **F** } *(opérateur **binaire** qui se lit " et ")*

| : { **V**, **F** } x { **V**, **F** } → { **V**, **F** } *(opérateur **binaire** qui se lit " ou ")*

! : { **V**, **F** } → { **V**, **F** } *(opérateur **unaire** qui se lit " non ")*

^ : { **V**, **F** } x { **V**, **F** } → { **V**, **F** } *(opérateur **binaire** qui se lit " ou exclusif ")*

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

p	q	! p	p ^ q	p & q	p | q
V	V	F	F	V	V
V	F	F	V	F	V
F	V	V	V	F	V
F	F	V	F	F	F

Remarque :

$\forall p \in \{ \mathbf{V}, \mathbf{F} \}, \forall q \in \{ \mathbf{V}, \mathbf{F} \}$, p **&**q est toujours évalué en entier *(p et q sont toujours évalués).*

$\forall p \in \{ \mathbf{V}, \mathbf{F} \}, \forall q \in \{ \mathbf{V}, \mathbf{F} \}$, p |q est toujours évalué en entier *(p et q sont toujours évalués).*

Java dispose de 2 clones des opérateurs binaires **&** et | . Ce sont les opérateurs **&&** et || qui se différentient de leurs originaux **&** et | par leur mode d'exécution optimisé (application de théorèmes de l'algèbre de boole) :

L'opérateur et optimisé : &&

Théorème

$\forall q \in \{ \mathbf{V}, \mathbf{F} \}$, **F** **&**q = **F**

Donc si p est faux (p = **F**) , il est inutile d'évaluer q car l'expression p **&**q est fausse (p **&**q = **F**), comme l'opérateur **&** évalue toujours l'expression q, Java à des fins d'optimisation de la vitesse d'exécution du bytecode dans la machine virtuelle Java, propose un opérateur ou noté **&&** qui a **la même table de vérité** que l'opérateur **&** mais qui applique ce théorème.

$\forall p \in \{ \mathbf{V}, \mathbf{F} \}, \forall q \in \{ \mathbf{V}, \mathbf{F} \}$, p **&&**q = p **&**q

Mais dans p**&&**q , *q n'est évalué que si p = **V**.*

L'opérateur ou optimisé : | |

Théorème

$\forall q \in \{ \mathbf{V}, \mathbf{F} \}$, **V** |q = **V**

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

comme l'opérateur | évalue toujours l'expression q, Java à des fins d'optimisation de la vitesse d'exécution du bytecode dans la machine virtuelle Java, propose un opérateur ou noté || qui applique ce théorème et qui a la même table de vérité que l'opérateur | .

∀p ∈ { **V**, **F** } , ∀q∈ { **V**, **F** } , p ||q = p |q
Mais dans p||q , *q n'est évalué que si p* = ***F***.

En résumé:

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

Nous allons voir ci-après une autre utilisation des opérateurs **&** et | sur des variables ou des valeurs immédiates en tant qu'opérateur bit-level.

4 - Opérateurs bits level

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

Opérateur	priorité	action	exemples
~	1	**complémente** les bits	~a; ~(a-b); ~7 (unaire)
<<	4	**décalage** gauche	x << 3 ; (a+2) << k ; -5 << 2 ;
>>	4	**décalage** droite avec signe	x >> 3 ; (a+2) >> k ; -5 >> 2 ;
>>>	4	**décalage** droite sans signe	x >>>3 ; (a+2) >>> k ;-5 >>> 2 ;

&	7	**et** booléen bit à bit	x & 3 ; (a+2) & k ; -5 & 2 ;
^	8	**ou** exclusif xor bit à bit	x ^ 3 ; (a+2) ^ k ; -5 ^ 2 ;
|	9	**ou** booléen bit à bit	x | 3 ; (a+2) | k ; -5 | 2 ;

Les tables de vérité de opérateurs "&", " | " et celle du ou exclusif " ^ " au niveau du bit sont identiques aux tables de vérité booléennes (seule la valeur des constantes **V** et **F** change, **V** est remplacé par le bit **1** et **F** par le bit **0**)

Table de vérité des opérateurs bit level

p	q	~ p	p & q	p | q	p ^ q
1	1	0	1	1	0
1	0	0	0	1	1
0	1	1	0	1	1
0	0	1	0	0	0

*L'opérateur ou exclusif ^ fonctionne aussi sur des variables de type **booléen***

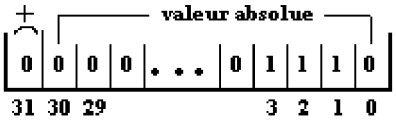
Afin de bien comprendre ces opérateurs, le lecteur doit bien connaître les différents codages des entiers en machine (binaire pur, binaire signé, complément à deux) car les entiers Java sont codés en complément à deux et la manipulation bit à bit nécessite une bonne compréhension de ce codage.

Afin que le lecteur se familiarise bien avec ces opérateurs de bas niveau nous détaillons un exemple pour **chacun d'entre eux**.

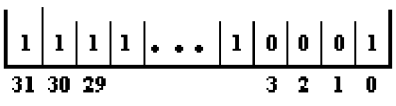
Les exemples en 4 instructions Java sur la même mémoire :

Rappel : **int** i = -14 ;

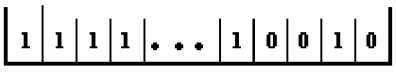
soit à représenter le nombre -14 dans la variable i de type **int** (entier signé sur 32 bits)



codage de |-14|= 14



complément à 1



addition de 1

Le nombre entier -14 s'écrit donc en complément à 2 : 1111..10010.

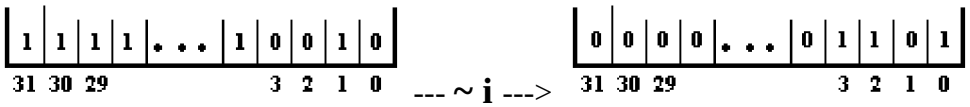
Soient la déclaration java suivante :

int i = -14 , j ;

Etudions les effets de chaque opérateur bit level sur cette mémoire i.

- Etude de l'instruction : j = ~ i

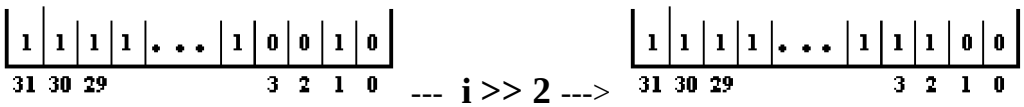
j = ~ i ; *// complémentation des bits de i*



Tous les bits 1 sont transformés en 0 et les bits 0 en 1, puis le résultat est stocké dans j qui contient la valeur 13 (car 000...01101 représente +13 en complément à deux).

- Etude de l'instruction : j = i >> 2

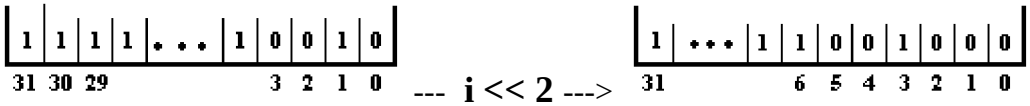
j = i >> 2 ; *// décalage avec signe de 2 bits vers la droite*



Tous les bits sont décalés de 2 positions vers la droite (vers le bit de poids faible), le bit de signe (ici 1) est recopié à partir de la gauche (à partir du bit de poids fort) dans les emplacements libérés (ici le bit 31 et le bit 30), puis le résultat est stocké dans j qui contient la valeur -4 (car 1111...11100 représente -4 en complément à deux).

- Etude de l'instruction : j = i << 2

j = i << 2 ; *// décalage de 2 bits vers la gauche*

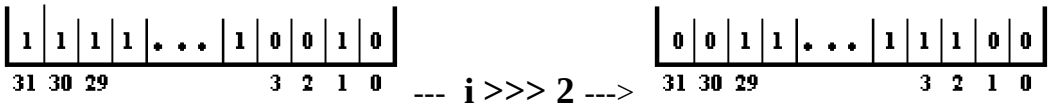


Tous les bits sont décalés de 2 positions vers la gauche (vers le bit de poids fort), des 0 sont introduits à partir de la droite (à partir du bit de poids faible) dans les emplacements libérés

(ici le bit 0 et le bit 1), puis le résultat est stocké dans j contient la valeur -56(car 11...1001000 représente -56 en complément à deux).

- Etude de l'instruction : j = i >>> 2

j = i >>> 2 ; *// décalage sans le signe de 2 bits vers la droite*



Instruction semblable au décalage >> mais au lieu de recopier le bit de signe dans les emplacements libérés à partir de la gauche, il y a introduction de 0 à partir de la gauche dans les 2 emplacements libérés (ici le bit 31 et le bit 30), puis le résultat est stocké dans j qui contient la valeur 1073741820.

En effet 0011...11100 représente 1073741820 en complément à deux :

$j = 2^{29} + 2^{28} + 2^{27} + ... + 2^3 + 2^2 = 2^2 . (2^{27} + 2^{26} + 2^{25} + ... + 2^1 + 1) = 2^2 . (2^{28} - 1) = 2^{30} - 2^2 = 1073741820)$

Exemples opérateurs arithmétiques

```
class AppliOperat_Arithm
{
    static void main(String[ ] args)
    {
        int x = 4, y = 8, z = 3, t = 7, calcul ;
        calcul = x * y - z + t ;
        System.out.println(" x * y - z + t = "+calcul);
        calcul = x * y - (z + t) ;
        System.out.println(" x * y - (z + t) = "+calcul);
        calcul = x * y % z + t ;
        System.out.println(" x * y % z + t = "+calcul);
        calcul = (( x * y) % z ) + t ;
        System.out.println("(( x * y) % z ) + t = "+calcul);
        calcul = x * y % ( z + t ) ;
        System.out.println(" x * y % ( z + t ) = "+calcul);
        calcul = x *(y % ( z + t ));
        System.out.println(" x *( y % ( z + t)) = "+calcul);
    }
}
```

Résultats d'exécution de ce programme :

x * y - z + t = 36

x * y - (z + t) = 22

x * y % z + t = 9

((x * y) % z) + t = 9

x * y % (z + t) = 2

x *(y % (z + t)) = 32

Exemples opérateurs booléens

```
class AppliOperat_Boole
{
    static void main(String[ ] args)
    {
        int x = 4, y = 8, z = 3, t = 7, calcul=0 ;
        boolean bool1 ;
        bool1 = x < y;
        System.out.println("  x < y  = "+bool1);
        bool1 = (x < y) & (z == t) ;
        System.out.println(" (x < y) & (z = = t)  = "+bool1);
        bool1 = (x < y) | (z == t) ;
        System.out.println(" (x < y) | (z = = t)  = "+bool1);
        bool1 = (x < y) && (z == t) ;
        System.out.println(" (x < y) && (z = = t)  = "+bool1);
        bool1 = (x < y) || (z == t) ;
        System.out.println(" (x < y) || (z = = t)  = "+bool1);
        bool1 = (x < y) || ((calcul=z) == t) ;
        System.out.println(" (x < y) || ((calcul=z) == t)  = "+bool1+"  ** calcul = "+calcul);
        bool1 = (x < y) | ((calcul=z) == t) ;
        System.out.println(" (x < y) | ((calcul=z) == t)  = "+bool1+"  ** calcul = "+calcul);
    }
}
```

Résultats d'exécution de ce programme :

x < y = true

(x < y) & (z = = t) = false

(x < y) | (z = = t) = true

(x < y) && (z = = t) = false

(x < y) || (z = = t) = true

(x < y) || ((calcul=z) == t) = true ** calcul = 0

(x < y) | ((calcul=z) == t) = true ** calcul = 3

Exemples opérateurs bit level

```
                        // OPERATEURS bit-Level

class AppliOperat_BitBoole
{
  static void main(String[ ] args)
  {
    int x, y, z ,t, calcul=0 ;
    x = 4;  // 00000100
    y = -5; // 11111011
    z = 3;  // 00000011
    t = 7;  // 00000111
    calcul = x & y ;
    System.out.println("  x & y  = "+calcul);
    calcul = x & z ;
    System.out.println("  x & z  = "+calcul);
    calcul = x & t ;
    System.out.println("  x & t  = "+calcul);
    calcul = y & z ;
    System.out.println("  y & z  = "+calcul);
    calcul = x | y ;
    System.out.println("  x | y  = "+calcul);
    calcul = x | z ;
    System.out.println("  x | z  = "+calcul);
    calcul = x | t ;
    System.out.println("  x | t  = "+calcul);
    calcul = y | z ;
    System.out.println("  y | z  = "+calcul);
    calcul = z ^ t ;
    System.out.println("  z ^ t  = "+calcul);
    System.out.println(" ~x  = "+~x+",  ~y  = "+~y+",  ~z  = "+~z+",  ~t  = "+~t);
  }
}
```

Résultats d'exécution de ce programme :

```
  x & y  = 0
  x & z  = 0
  x & t  = 4
  y & z  = 3
  x | y  = -1
  x | z  = 7
  x | t  = 7
  y | z  = -5
  z ^ t  = 4
 ~x  = -5,  ~y  = 4,  ~z  = -4,  ~t  = -8
```

Exemples opérateurs bit level - Décalages

```
class AppliOperat_BitDecalage
{
    static void main(String[ ] args)
    {
        int x,y,z, calcul = 0 ;
        x = -14; // 1...11110010
        y = x;
        z = x;
        calcul = x >>2;   // 1...11111100
        System.out.println(" x >>2  = "+calcul);
        calcul = y <<2 ; // 1...11001000
        System.out.println(" y <<2  = "+calcul);
        calcul = z >>>2 ; // 001...111100
        System.out.println(" z >>>2  = "+calcul);
    }
}
```

Résultats d'exécution de ce programme :

x >>2 = -4

y <<2 = -56

z >>>2 = 1073741820

Exemples opérateurs post - incrémentation / décrémentation

```
// OPERATEUR ARITHMETIQUE   x--   post décrémentation

class AppliOperat_PostDecr
{
   static void main(String[ ] args)
   {
      int x = 4, y = 8, z = 3, t = 7, calcul ;
      calcul = x-- ;
      System.out.println(" x  = "+x+"  x--  = "+calcul);
      calcul = y---1 ;
      System.out.println(" y  = "+y+"  y---1  = "+calcul);
      calcul = z-- - z ;
      System.out.println(" z  = "+z+"  z-- - z  = "+calcul);
      calcul = z - z-- ;
      System.out.println(" z  = "+z+"  z - z--  = "+calcul);
   }
}

/*Résultats :
     x  = 3  x--  = 4
     y  = 7  y---1  = 7
     z  = 2  z-- - z  = 1
     z  = 1  z - z--  = 0
*/
```

```
                // OPERATEUR ARITHMETIQUE  x++   post incrémentation

class AppliOperat_PostIncr
{
   static void main(String[ ] args)
   {
      int x = 4, y = 8, z = 3, t = 7, calcul ;
      calcul = x++ ;
      System.out.println(" x  = "+x+"  x++  = "+calcul);
      calcul = y+++1 ;
      System.out.println(" y  = "+y+"  y+++1  = "+calcul);
      calcul = z++ - z ;
      System.out.println(" z  = "+z+"  z++ - z  = "+calcul);
      calcul = z - z++ ;
      System.out.println(" z  = "+z+"  z - z++  = "+calcul);
   }
}

/*Résultats :
     x  = 5  x++  = 4
     y  = 9  y+++1  = 9
     z  = 4  z++ - z  = -1
     z  = 5  z - z++  = 0
*/
```

Les instructions
Java 2

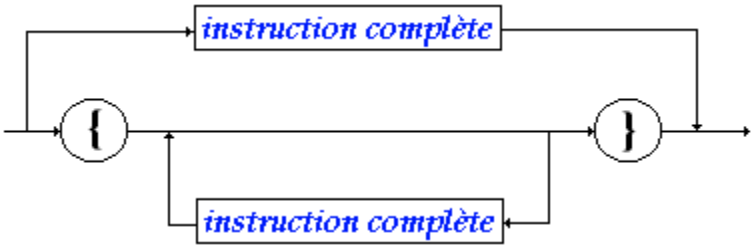
1 - les instructions de bloc

Une large partie de la norme ANSI du langage C est reprise dans Java.

- Les commentaires sur une ligne débutent par **//....** (spécifique Java)
- Les commentaires sur plusieurs lignes sont encadrés par **/* ... */** (vient du C)

Ici, nous expliquons les instructions Java en les comparant à pascal-delphi. Voici la syntaxe d'une instruction en Java :

instruction :

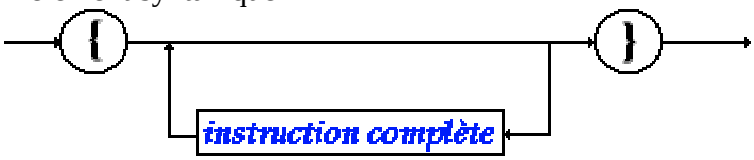


instruction complète :

Toutes les instructions se terminent donc en Java par un point-virgule " ; "

bloc - instruction composée :

L'élément syntaxique



est aussi dénommé **bloc** ou **instruction composée** au sens de la **visibilité** des variables Java.

visibilité dans un bloc - instruction :

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

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

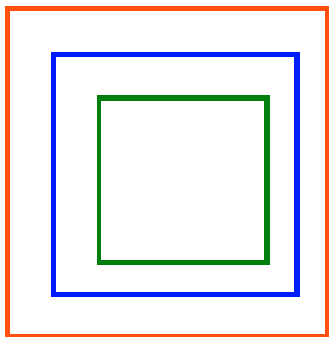


schéma d'imbrication des 3 blocs

Nous examinons ci-dessous l'ensemble des **instructions simples** de Java.

2 - l'affectation

Java est un langage de la famille des langages hybrides, il possède la notion d'instruction d'affectation.

Le symbole d'affectation en Java est " = ", soit par exemple :

```
x = y ;
// x doit obligatoirement être un identificateur de variable.
```

Affectation simple

L'affectation peut être utilisée dans une expression :

soient les instruction suivantes :

```
int a , b = 56 ;
a = (b = 12)+8 ;  // b prend une nouvelle valeur dans l'expression
a = b = c = d =8 ; // affectation multiple
```

simulation d'exécution Java :

instruction	*valeur de a*	*valeur de b*
int a , b = 56 ;	**a = ???**	**b = 56**

a = (b = 12)+8 ;	**a = 20**	**b = 12**

3 - Raccourcis et opérateurs d'affectation

Soit **op** un opérateur appartenant à l'ensemble des opérateurs suivant

{ +, - , * , / , % , << , >> , >>> , & , | , ^ },

Il est possible d'utiliser sur une seule variable le nouvel opérateur **op=** construit avec l'opérateur **op**.

Il s'agit plus d'un **raccourci syntaxique** que d'un opérateur nouveau (seule sa traduction en bytecode diffère : la traduction de a **op**= b devrait être plus courte en instructions p-code que a = a **op** b, bien que les optimiseurs soient capables de fournir le même code optimisé).

x **op**= y ; signifie en fait : x = x **op** y

Soient les instruction suivantes :

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

simulation d'exécution Java :

instruction	*valeur de a*	*valeur de b*
int a , b = 56 ;	**a = ???**	**b = 56**
a = -8 ;	**a = -8**	**b = 56**
a += b ;	**a = 48**	**b = 56**
b *= 3 ;	**a = 48**	**b = 168**

Remarques :

- ***Cas d'une optimisation intéressante dans l'instruction suivante :***
  table[f(a)] = table[f(a)] + x ; *// où f(a) est un appel à la fonction f qui serait longue à calculer.*

- ***Si l'on réécrit l'instruction précédente avec l'opérateur += :***
  table[f(a)] += x ; *// l'appel à f(a) n'est effectué qu'une seule fois*

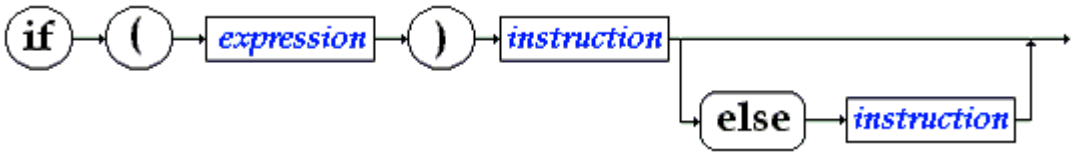
Les instructions conditionnelles

Java 2

---

1 - l'instruction conditionnelle

Syntaxe :



Schématiquement les conditions sont de deux sortes :

- **if** (Expr) Instr ;
- **if** (Expr) Instr ; **else** Instr ;

La définition de l'instruction conditionnelle de java est classiquement celle des langages algorithmiques; comme en pascal l'expression doit être de type booléen (différent du C), la notion d'instruction a été définie plus haut.

Exemple d'utilisation du if..else (comparaison avec pascal)

Pascal-Delphi	Java

var a , b , c : integer ;
....
if b=0 **then** c := 1
else begin
c := a / b;
writeln("c = ",c);
end;
c := a*b ;
if c <>0 **then** c:= c+b
else c := a
	int a , b , c ;
	....
	if (b = = 0) c =1 ;
	else {
	c = a / b;
	System.out.println("c = " + c);
	}
	if ((c = a*b) != 0) c += b;
	else c = a;

Remarques :

- L'instruction " **if** ((c = a*b) != 0) c +=b; **else** c = a; " contient une affectation intégrée dans le test afin de vous montrer les possibilités de Java : la valeur de a*b est rangée dans c avant d'effectuer le test sur c.

- Comme pascal, Java contient le manque de fermeture des instructions conditionnelles ce qui engendre le classique problème du dandling **else** d'algol, c'est le compilateur qui résout l'ambiguïté par rattachement du **else** au dernier **if** rencontré (évaluation par la gauche).

  L'instruction suivante est ambiguë :

  if (Expr1) **if** (Expr2) InstrA ; **else** InstrB ;

  Le compilateur résout l'ambigüité de cette instruction ainsi (rattachement du **else** au dernier **if**):

  if (Expr1) **if** (Expr2) InstrA ; **else** InstrB ;

- Comme en pascal, si l'on veut que l'instruction « **else** InstrB ; » soit rattachée au premier **if**, il est nécessaire de parenthéser (introduire un bloc) le second **if** :

Exemple de parenthésage du else pendant

Pascal-Delphi	Java

if Expr1 **then**
begin
  if Expr2 **then** InstrA
end
else InstrB
	if (Expr1) **{**
	  if (Expr2) InstrA ;
	}
	else InstrB

2 - l'opérateur conditionnel

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

Syntaxe :

Où ***expression*** renvoie une valeur booléenne (le test), les deux termes ***valeur*** sont des expressions générales (variable, expression numérique, booléenne etc...) renvoyant une valeur de type quelconque.

Sémantique :

Exemple :

```
int a,b,c ;
c =   a = = 0  ?  b  :  a+1 ;
```

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

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

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

---

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

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

---

question : que fait ce morceau le programme ci-après ?

```
int a , b , c ; ....
c = a>b ? (b=a) : (a=b) ;
```

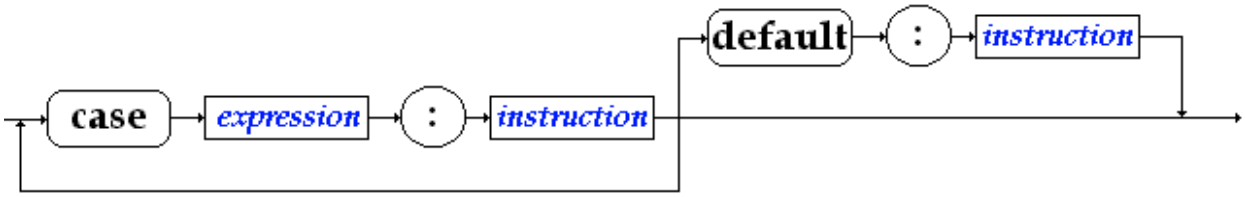
réponse :
a,b,c contiennent après exécution le plus grand des deux entiers contenus au départ dans a et b.

3 - l'opérateur switch...case

Syntaxe :

switch :

bloc switch :



Sémantique :

- La partie expression d'une instruction switch doit être une expression ou une variable du type **byte**, **char**, **int** ou bien **short**.
- La partie expression d'un bloc switch doit être une constante ou une valeur immédiate du type **byte**, **char**, **int** ou bien **short**.
- **switch** <Epr1> s'appelle la partie sélection de l'instruction : il y a évaluation de <Epr1> puis selon la valeur obtenue le programme s'exécute en séquence à partir du case contenant la valeur immédiate égal. Il s'agit donc d'un déroutement du programme dès que <Epr1> est évaluée vers l'instruction étiquetée par le case <Epr1> associé.

Exemples :

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

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

Utilisée telle quelle, cette instruction n'est pas structurée et donc son utilisation est déconseillée sous cette forme. Par contre elle est très souvent utilisée avec l'instruction **break** afin de simuler le comportement de l'instruction structurée **case**..**of** du pascal (ci-dessous deux écritures équivalentes) :

Exemple de switch..case..break

Pascal-Delphi	Java
var x : **char** ; **....** **case** x of 'a' : InstrA; 'b' : InstrB; **else** InstrElse **end**;	**char** x ; **....** **switch** (x) **{** **case** 'a' : InstrA ; break; **case** 'b' : InstrB ; break; **default** : InstrElse; **}**

Dans ce cas le déroulement de l'instruction **switch** après déroutement vers le bon **case**, est interrompu par le **break** qui renvoie la suite de l'exécution après la fin du **bloc switch**. Une telle utilisation correspond à une utilisation de if...else imbriqués (donc une utilisation structurée) mais devient plus lisible que les **if ..else** imbriqués, elle est donc fortement conseillée dans ce cas.

En reprenant les deux exemples précédents :

Exemples :

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

Il est toujours possible d'utiliser des instructions **if** … **else** imbriquées pour représenter un switch structuré (switch avec break) :

Programmes équivalents switch et if...else :

Java - switch	Java - if...else
int x = 10; **switch** (x+1) { **case** 11 : System.out.println(">> case 11"); break; **case** 12 : System.out.println(">> case 12"); break; **default** : System.out.println(">> default"); }	**int** x = 10; **if** (x+1= = 11)System.out.println(">> case 11"); **else** **if** (x+1= = 12)System.out.println(">> case 12"); **else** System.out.println(">> default");

Nous conseillons au lecteur de n'utiliser le switch qu'avec des break afin de bénéficier de la structuration.

Exemples instruction et opérateur conditionnels

```
            // INSTRUCTION CONDITIONNELLE       if ... else

class AppliInstruction_cond {

  public static void main(String[] args){
    int x = Readln.unint();
    String strl;

   /* 3 versions du positionnement d'un nombre entré
      au clavier, par rapport aux nombres 2 et 8 */

    if(x>2)
      if(x<=8) strl ="x entre ]2,8]";
       else strl ="x entre ]8, Max]";
    else strl="x entre [min,2]";
    System.out.println(strl);

    strl ="x entre [min,2]";
    if(x>2)
      if(x<=8) strl ="x entre ]2,8]";
       else strl ="x entre ]8, Max]";
    System.out.println(strl);

    strl ="x entre ]8, Max]";
    if(x>2){
      if(x<=8) strl ="x entre ]2,8]";
      }
    else strl ="x entre [min,2]";
    System.out.println(strl);
  }

}
```

```
            // OPERATEUR CONDITIONNEL       < ... ? ... : ... >

class AppliOpererateur_cond {

  public static void main(String[] args){
    int x = 14;
    String strl, str2, str3, str4;
    boolean Ok=true;

    strl = (x<8) ? "x inférieur à 8" : "x supérieur ou égal à 8";
    str2 = (x<=4) ? "x inférieur à 4" : "x supérieur ou égal à 4";
    str3 = ((x>10) | (Ok=false)) ? "(x>10) | (Ok=false)" : "non [(x>10) | (Ok=false)]";
    str4 = ( (Ok=false) & (x>10) ) ? "(Ok=false) & (x>10)" : "non [(Ok=false) & (x>10)]";
    System.out.println(strl+"\n"+str2+"\n"+str3+"\n"+str4);
  }

}
```

Exemples instruction et opérateur conditionnels

```
        /* Le max de 3 entiers avec l'opérateur conditionnel
           ... ? ... : ...
        */

class AppliMax3operCond
{
  static void main(String[ ] args)
  {
    int x, y, z ;
    System.out.print("x = ");
    x = Readln.unint();
    System.out.print("y = ");
    y = Readln.unint();
    System.out.print("z = ");
    z = Readln.unint();
    int max;
    max = z > (max = (x<y) ? y : x) ? z : max;
    System.out.println("Le maximum = "+max);
  }
}
```

```
        /* Le max de 3 entiers avec l'instruction conditionnelle
                    if ... else
        */

class AppliMax3InstrCond
{
  static void main(String[ ] args)
  {
    int x, y, z ;
    System.out.print("x = ");
    x = Readln.unint();
    System.out.print("y = ");
    y = Readln.unint();
    System.out.print("z = ");
    z = Readln.unint();
    int max = 0;
    if( (x>=y) & (x>=z)) max = x;
    else
      if ((y>=x) & (y>=z)) max = y;
      else
        if ((z>=x) & (z>=y)) max = z;
        else System.out.println("Impossible");
    System.out.println("Le maximum = "+max);
  }
}
```

Exemple switch … case , avec et sans break

```
class AppliInstruction_Switch1 {

  public static void main(String[] args){
    final int Y = 10; // une constante en Java est précédée du mot clef "final"

   /* essayez les nombres x=10 puis x=11, puis x=12
    et comparez les deux comportements */
    int x = 10, a=x;

   // sans l'instruction break :
    switch (x)
    {
      case Y : x++;
      case Y+1 : x++;
      case Y+2 : x++;
      default : x++;
    }
    System.out.println("x = "+x);

   // avec l'instruction break :
    switch (a)
    {
      case Y : a++;
        break;
      case Y+1 : a++;
        break;
      case Y+2 : a++;
        break;
      default : a++;
    }
    System.out.println("a = "+a);
  }
}
```

Résultats d'exécution du programme pour x = 10 :

x = 14

a = 11

Résultats d'exécution du programme pour x = 11 :

x = 14

a = 12

Résultats d'exécution du programme pour x = 12 :

x = 14

a = 13

Les instructions itératives
Java 2

1 - l'instruction while

Syntaxe :

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

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

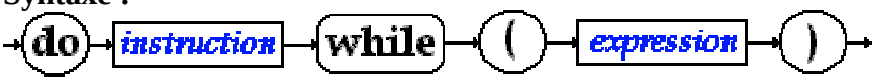
Exemple de boucle while

Pascal-Delphi	Java
while Expr **do** Instr	**while (** Expr **)** Instr ;

while Expr **do**
begin
InstrA ;
InstrB ; ...
end
	while (Expr **)**
	{
	InstrA ;
	InstrB ; ...
	}

2 - l'instruction do ... while

Syntaxe :



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

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

Sa sémantique peut aussi être expliquée à l'aide d'une autre instruction Java **while** :

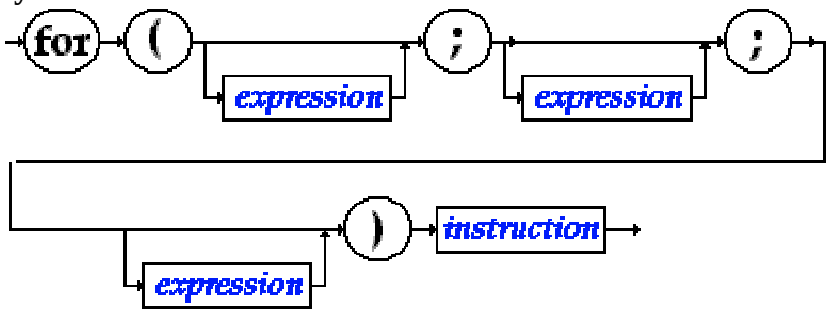
do Instr **while** (Expr) <=> Instr ; **while** (Expr) Instr

Exemple de boucle do...while

Pascal-Delphi	Java
repeat InstrA ; InstrB ; ... **until not** Expr	**do** **{** InstrA ; InstrB ; ... **} while (** Expr **)**

3 - l'instruction for(...)

Syntaxe :



Sémantique :
Une boucle **for** contient 3 expressions **for** (**Expr1** ; **Expr2** ; **Expr3**) Instr, d'une manière générale chacune de ces expressions joue un rôle différent dans l'instruction **for.** Une instruction for en Java (comme en C) est plus puissante et plus riche qu'une boucle **for** dans d'autres langages algorithmiques. Nous donnons ci-après une sémantique minimale :

- **Expr1** sert à initialiser une ou plusieurs variables (dont éventuellement la variable de contrôle de la boucle) sous forme d'une liste d'instructions d'initialisation séparées par des virgules.
- **Expr2** sert à donner la condition de rebouclage sous la forme d'une expression renvoyant une valeur booléenne (le test de l'itération).
- **Expr3** sert à réactualiser les variables (dont éventuellement la variable de contrôle de la boucle)sous forme d'une liste d'instructions séparées par des virgules.

L'instruction "**for** (**Expr1** ; **Expr2** ; **Expr3**) Instr" fonctionne au minimum comme l'instruction algorithmique **pour**... **fpour**, elle est toutefois plus puissante que cette dernière.

Sa sémantique peut aussi être approximativement**(*)** expliquée à l'aide d'une autre instruction Java **while** :

for (**Expr1** ; **Expr2** ; **Expr3**) Instr	**Expr1** ; **while** (**Expr2**) { Instr ; **Expr3** }

(*)Nous verrons au paragraphe consacré à l'instruction **continue** que si l'instruction **for** contient un **continue** cette définition sémantique n'est pas valide.

Exemples montrant la puissance du for

Pascal-Delphi	Java

for i:=1 **to** 10 **do**
begin
InstrA ;
InstrB ; ...
end
	for (**i = 1**; **i<=10**; **i++**)
	{
	InstrA ;
	InstrB ; ...
	}

i := 10; k := i;
while (i>-450) **do**
begin
InstrA ;
InstrB ; ...
k := k+i;
i := i-15;
end
	for (**i = 10, k = i** ;**i>-450** ; **k += i , i -= 15**)
	{
	InstrA ;
	InstrB ; ...
	}

i := n;
while i<>1 **do**
if i **mod** 2 = 0 **then** i := i **div** 2
else i := i+1
	for (**i = n** ;**i !=1** ; **i % 2 == 0 ? i /=2 : i++**);
	// pas de corps de boucle !

- Le premier exemple montre une boucle for classique avec la variable de contrôle "i" (indice de boucle), sa borne initiale "i=1" et sa borne finale "10", le pas d'incrémentation séquentiel étant de 1.
- Le second exemple montre une boucle toujours contrôlée par une variable "i", mais dont le pas de décrémentation séquentiel est de -15.
- Le troisième exemple montre une boucle aussi contrôlée par une variable "i", mais dont la variation n'est pas séquentielle puisque la valeur de i est modifiée selon sa parité (**i % 2 == 0 ? i /=2 : i++**).

Voici un exemple de boucle **for** dite **boucle infinie** :

for (; ;); est équivalente à **while** (true);

Voici une boucle ne possédant pas de variable de contrôle(f(x) est une fonction déjà déclarée) :

```
for (int n=0 ; Math.abs(x-y) < eps ; x = f(x) );
```

Terminons par une boucle for possédant deux variables de contrôle :

```
//inverse d'une suite de caractère dans un tableau par permutation des deux extrêmes
char [ ] Tablecar ={'a','b','c','d','e','f'} ;
for ( i = 0 , j = 5 ;  i<j ; i++ , j-- )
{ char car ;
   car = Tablecar[i];
   Tablecar[i ]= Tablecar[j];
   Tablecar[j] = car;
  }
```

dans cette dernière boucle ce sont les variations de i et de j qui contrôlent la boucle.

Remarques récapitulatives sur la boucle for en Java :

rien n'oblige à incrémenter ou décrémenter la variable de contrôle,

rien n'oblige à avoir une instruction à exécuter (corps de boucle),

rien n'oblige à avoir une variable de contrôle,

rien n'oblige à n'avoir qu'une seule variable de contrôle.

Exemples boucles for , while , do…while

```
class Appliboucle_for {

  public static void main(String[] args){
    for (int x=0 ; x<10 ; x++){
      System.out.println("x = " + x);
    }
  }
}
```

```
class Appliboucle_while {

  public static void main(String[] args){
    int x = 0;
    while (x<10){
      System.out.println("x = " + x);
      x++;
    }
  }

}
```

```
class Appliboucle_do_while {

  public static void main(String[] args){
    int x = 0;
    do{
      System.out.println("x = " + x);
      x++;
    }
    while (x<10);
  }
}
```

Résultats d'exécution des ces 3 programmes :

x = 0

x = 1

….

x = 9

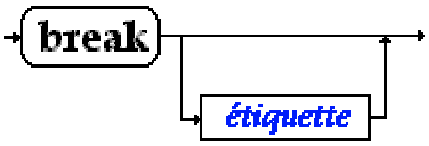
Les instructions de rupture de séquence

Java 2

---

1 - l'instruction d'interruption break

Syntaxe :



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

Lorsque **break** est présente dans l'une des trois itérations **while**, **do..while**, **for** :

- Si **break** n'est pas suivi d'une étiquette, elle interrompt l'exécution de la boucle dans laquelle elle se trouve, l'exécution se poursuit après le corps d'instruction.
- Si **break** est suivi d'une étiquette, elle fonctionne comme un **goto** (utilisation **déconseillée** en programmation moderne sauf pour le **switch**, c'est pourquoi nous n'en dirons pas plus pour les boucles !)

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

```
int [ ] table = {12,-5,7,8,-6,6,4,78,2};
int elt = 4;
for ( i = 0 ; i<8 ; i++ )
  if (elt= =table[i]) break ;
if (i = = 8)System.out.println("valeur : "+elt+" pas trouvée.");
else System.out.println("valeur : "+elt+" trouvée au rang :"+i);
```

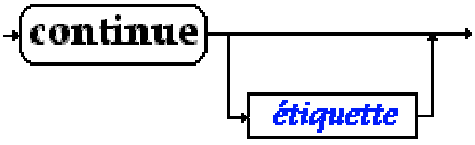
Explications

Si la valeur de la variable elt est présente dans le tableau table **if** (elt= =table[i]) est true et **break** est exécutée (arrêt de la boucle et exécution de **if** (i = = 8)...).

Après l'exécution de la boucle **for**, lorsque l'instruction **if** (i = = 8)... est exécutée, soit la boucle s'est exécutée complètement (recherche infructueuse), soit le **break** l'a arrêtée prématurément (elt trouvé dans le tableau).

2 - l'instruction de rebouclage continue

Syntaxe :



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

Lorsque **continue** est présente dans l'une des trois itérations **while**, **do..while**, **for** :

- Si **continue** n'est pas suivi d'une étiquette elle interrompt l'exécution de la séquence des instructions situées après elle, l'exécution par rebouclage de la boucle. Elle agit comme si l'on venait d'exécuter la dernière instructions du corps de la boucle.
- Si **continue** est suivi d'une étiquette elle fonctionne comme un **goto** (utilisation **déconseillée** en programmation moderne, c'est pourquoi nous n'en dirons pas plus !)

Exemple d'utilisation du continue dans un for :

```
int [ ] ta = {12,-5,7,8,-6,6,4,78,2}, tb = new int[8];
for ( i = 0, n = 0 ; i<8 ;  i++ , k = 2*n )
{ if ( ta[i] = = 0 ) continue ;
  tb[n] = ta[i];
  n++;
}
```

Explications

Rappelons qu'un **for** s'écrit généralement :
for (**Expr1** ; **Expr2** ; **Expr3**) Instr
L'instruction **continue** présente dans une telle boucle **for** s'effectue ainsi :

- exécution immédiate de **Expr3**
- ensuite, exécution de **Expr2**
- reprise de l'exécution du corps de boucle.

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

Le déroulement est alors le suivant :

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

et la boucle se poursuit en fonction de la valeur de la condition de rebouclage.

Cette boucle recopie dans le tableau d'entiers **tb** les valeurs non nulles du tableau d'entiers **ta**.

Attention

Nous avons déjà signalé plus haut que l'équivalence suivante entre un **for** et un **while**

for (**Expr1** ; **Expr2** ; **Expr3**) Instr	**Expr1** ; **while** (**Expr2**) { Instr ; **Expr3** }

valide dans le cas général, était mise en défaut si le corps d'instruction contenait un **continue**.

Voyons ce qu'il en est en reprenant l'exemple précédent. Essayons d'écrire la boucle **while** qui lui serait équivalente selon la définition générale. Voici ce que l'on obtiendrait :

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

Dans le **while** le **continue** réexécute la condition de rebouclage **i<8** sans exécuter l'expression **i++ ; k = 2*n;** (nous avons d'ailleurs ici une boucle infinie).

Une boucle **while** strictement équivalente au **for** précédent pourrait être la suivante :

for (**i = 0, n = 0** ; **i<8** ; **i++ , k = 2*n**)
{ **if** (ta[i] = = 0) **continue** ;
  tb[n] = ta[i];
  n++;
	i = 0; n = 0 ;
	while (**i<8**)
	{ **if** (ta[i] = = 0)
	  i++ ; k = 2*n;

}	**continue** ;
	}
	tb[n] = ta[i];
	n++;
	i++ ; k = 2*n;
	}

Exemples break dans une boucle for , while , do…while

```
// BREAK dans un DO...WHILE

class AppliBreak_do_while {

  public static void main(String[] args){
    int x = 0;
    do{
      System.out.println("x = " + x);
      if(x / 5 == 3)
        break;
      x++;
    }
    while (x<50);
  }
}
```

```
// BREAK dans un FOR

class AppliBreak_for {

  public static void main(String[] args){
    for (int x=0 ; x<50 ; x++){
      if(x / 5 == 3)
        break;
      System.out.println("x = " + x);
    }
  }
}
```

```
// BREAK dans un WHILE

class AppliBreak_while {

  public static void main(String[] args){
    int x = 0;
    while (x<50){
      System.out.println("x = " + x);
      if(x / 5 == 3)
        break;
      x++;
    }
  }
}
```

Résultats d'exécution des ces 3 programmes :

x = 0

x = 1

….

x = 15

Exemples continue dans une boucle for , while , do…while

```
// CONTINUE dans un DO...WHILE

class AppliContinue_do_while {

    public static void main(String[] args){
        int x = 0;
        do{
            x++;
            if(x %2 ==  0)
                continue; // affiche les impairs seulement
            System.out.println("x = " + x);
        }while (x<50);
    }
}
```

```
// CONTINUE dans un FOR

class AppliContinue_for {

    public static void main(String[] args){
        for (int x=0 ; x<50 ; x++){
            if(x %2 ==  0)
                continue; // affiche les impairs seulement
            System.out.println("x = " + x);
        }
    }
}
```

```
// CONTINUE dans un WHILE

class AppliContinue_while {

    public static void main(String[] args){
        int x = 0;
        while (x<50){
            x++;
            if(x %2 ==  0)
                continue; // affiche les impairs seulement
            System.out.println("x = " + x);
        }
    }

}
```

Résultats d'exécution des ces 3 programmes :

x = 1

x = 3

…. *(les entiers impairs jusqu'à 49)*

x = 49

classe avec méthode static

Java2

Une classe suffit
Les méthodes sont des fonctions
Transmission des paramètres en Java
Visibilité des variables

Avant d'utiliser les possibilités offertes par les classes et les objets en Java, apprenons à utiliser et exécuter des applications simples Java ne nécessitant pas la construction de nouveaux objets, ni de navigateur pour s'exécuter.

Comme Java est un langage orienté objet, un programme Java est composé de plusieurs classes, nous nous limiterons à une seule classe.

1 - Une classe suffit

On peut très grossièrement assimiler un programme Java ne possédant qu'une seule classe, à un programme principal classique d'un langage de programmation algorithmique.

- Une classe minimale commence obligatoirement par le mot **class** suivi de l'identificateur de la classe puis du corps d'implémentation de la classe dénommé **bloc de classe**.
- Le bloc de classe est parenthésé par deux accolades "**{**" et "**}**".

Syntaxe d'une classe exécutable

Exemple1 de classe minimale :

```
class Exemple1 {  }
```

Cette classe ne fait rien et ne produit rien.

Comme en fait, une classe quelconque peut s'exécuter toute seule à condition qu'elle possède dans ses déclarations internes la méthode **main** qui sert à lancer l'exécution de la classe (fonctionnement semblable au lancement du programme principal).

Exemple2 de squelette d'une classe minimale exécutable :

```
class Exemple2
{
  public static void main(String[ ] args)
  { // c'est ici que vous écrivez votre programme principal
   }
}
```

Exemple3 trivial d'une classe minimale exécutable :

```
class Exemple3
{
  public static void main(String[ ] args)
  { System.out.println("Bonjour !");
   }
}
```

Exemples d'applications à une seule classe

Nous reprenons deux exemples de programme utilisant la boucle **for**, déjà donnés au chapitre sur les instructions, cette fois-ci nous les réécrirons sous la forme d'une application exécutable.

Exemple1

```
class Application1
{
  public static void main(String[ ] args)
  { /* inverse d'une suite de caractère dans un tableau par
       permutation des deux extrêmes */
    char [ ] Tablecar ={'a','b','c','d','e','f'} ;
    int i, j ;
    System.out.println("tableau avant : " + String.valueOf(Tablecar));
    for ( i = 0 , j = 5 ;  i<j ; i++ , j-- )
     { char car ;
       car = Tablecar[i];
       Tablecar[i ]= Tablecar[j];
       Tablecar[j] = car;
     }
    System.out.println("tableau après : " + String.valueOf(Tablecar));
  }
}
```

Il est impératif de sauvegarder la classe dans un fichier qui **porte le même nom** (majuscules et minuscules inclues) ici "**Application1**.**java**". Lorsque l'on demande la compilation (production du bytecode) de ce fichier source "**Application1**.**java**" le fichier cible produit en bytecode se dénomme "**Application1**.**class**", il est alors prêt à être interprété par une

machine virtuelle java.

Le résultat de l'exécution de ce programme est le suivant :

```
tableau avant : abcdef
tableau après : fedcba
```

Exemple2

```
class Application2
{
  public static void main(String[ ] args)
   { // recherche séquentielle dans un tableau
     int [ ] table= {12,-5,7,8,-6,6,4,78};
     int elt = 4, i ;
     for ( i = 0 ; i<8 ; i++ )
       if (elt= =table[i]) break ;
     if (i = = 8) System.out.println("valeur : "+elt+" pas trouvée.");
     else System.out.println("valeur : "+elt+" trouvée au rang :"+i);
   }
}
```

Après avoir sauvegardé la classe dans un fichier qui **porte le même nom** (majuscules et minuscules incluses) ici "**Application2.java**", la compilation de ce fichier "**Application2.java**" produit le fichier "**Application2.class**" prêt à être interprété par notre machine virtuelle java.

Le résultat de l'exécution de ce programme est le suivant :

```
valeur : 4 trouvée au rang :6
```

Conseil de travail :

Reprenez tous les exemples simples du chapitre sur les instructions de boucle et le switch en les intégrant dans une seule classe (comme nous venons de le faire avec les deux exemples précédents) et exécutez votre programme.

2 - Les méthodes sont des fonctions

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

Méthode élémentaire de classe

Bien que Java distingue deux sortes de méthodes : les **méthodes de classe** et les **méthodes d'instance**, pour l'instant dans cette première partie nous décidons à titre pédagogique et simplificateur de n'utiliser que **les méthodes de classe**, l'étude du chapitre sur Java et la programmation orientée objet apportera les compléments adéquats sur les méthodes d'instances.

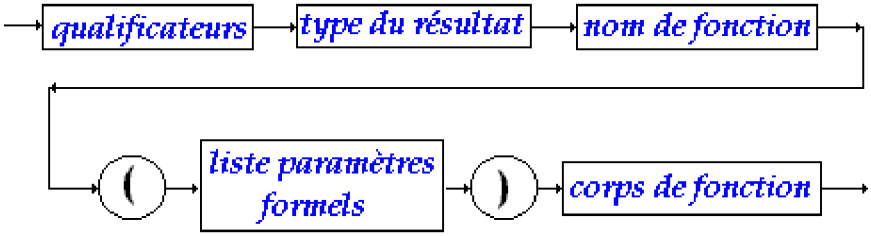
Une méthode de classe commence **obligatoirement** par le mot clef **static**.

Donc par la suite dans ce chapitre, lorsque nous emploierons le mot méthode sans autre adjectif, il s'agira d'une **méthode de classe**, comme nos application ne possèdent qu'une seule classe, nous pouvons assimiler ces méthodes aux fonctions de l'application et ainsi retrouver une *utilisation classique de Java en mode application*.

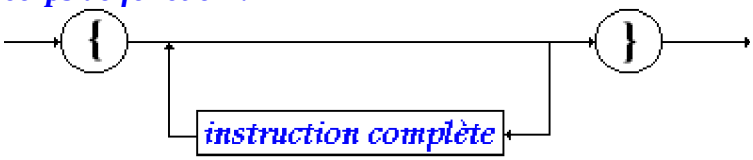
Déclaration d'une méthode

La notion de fonction en Java est semblable à celle du C et à Delphi, elle comporte **une en-tête** avec des paramètres formels **et un corps de fonction** ou de méthode qui contient les instructions de la méthode qui seront exécutées lors de son appel. La déclaration et l'implémentation doivent être consécutives comme l'indique la syntaxe ci-dessous :

Syntaxe :



corps de fonction :



Nous dénommons en-tête de fonction la partie suivante :
<qualificateurs><type du résultat><nom de fonction> **(**<liste paramètres formels>**)**

Sémantique :

- Les qualificateurs sont des mots clefs permettant de modifier la ***visibilité*** ou le ***fonctionnement*** d'une méthode, nous n'en utiliserons pour l'instant qu'un seul : le mot clef **static** permettant de désigner la méthode qu'il qualifie comme une méthode de classe dans la classe où elle est déclarée. Une méthode n'est pas nécessairement qualifiée donc ce mot clef peut être omis.

- Une méthode peut renvoyer un résultat d'un type Java quelconque en particulier d'un des types élémentaires (**int**, **byte**, **short**, **long**, **boolean**, **double**, **float**, **char**) et nous verrons plus loin qu'elle peut renvoyer un résultat de type objet comme en Delphi. Ce mot clef ne doit pas être omis.

- Il existe en Java comme en C une écriture fonctionnelle correspondant aux procédures des langages procéduraux : on utilise une **fonction qui ne renvoie aucun résultat**. L'approche est inverse à celle du pascal où la procédure est le bloc fonctionnel de base et la fonction n'en est qu'un cas particulier. En Java la fonction (ou méthode) est le seul bloc fonctionnel de base et la procédure n'est qu'un cas particulier de fonction dont le retour est de type **void**.

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

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

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

int calculer( ){.....}	renvoie un entier de type **int**
boolean tester( ){.....}	renvoie un entier de type **boolean**
void uncalcul( ){.....}	procédure **ne renvoyant rien**

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

int calculer(**byte** a, **byte** b, **int** x) {.....}	**fonction à 3 paramètres**
boolean tester(**int** k) {.....}	**fonction à 1 paramètre**
void uncalcul(**int** x, **int** y, **int** z) {.....}	**procédure à 3 paramètres**

Appel d'une méthode

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

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

```
class Application3
    {
      public static void main(String[ ] args)
       {
            afficher( );
       }
      static void afficher( )
     {
        System.out.println("Bonjour");
      }
    }
```

Appel de la méthode afficher

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

```
class Application4
{  public static void main(String[ ] args)
    { // recherche séquentielle dans un tableau
       int [ ] table= {12,-5,7,8,-6,6,4,78};
       long elt = 4;
       int i ;
       for ( i = 0 ; i<=8 ; i++ )
       if (elt= =table[i]) break ;
       afficher(i,elt);
     }
  static void afficher (int rang , long val)
  { if (rang == 8)
       System.out.println("valeur : "+val+" pas
trouvée.");
      else
       System.out.println("valeur : "+val+" trouvée
au rang :"+ rang);
   }
}
```

Appel de la méthode afficher

```
afficher(i,elt);
```

Les deux paramètres effectifs "**i**" et "**elt**" sont du même type que le paramètre formel associé.

- Le paramètre effectif "**i**" est associé au paramètre formel **rang**.

- Le paramètre effectif "**elt**" est associé au paramètre formel **val**.

3 - Transmission des paramètres

Rappelons tout d'abord quelques principes de base :

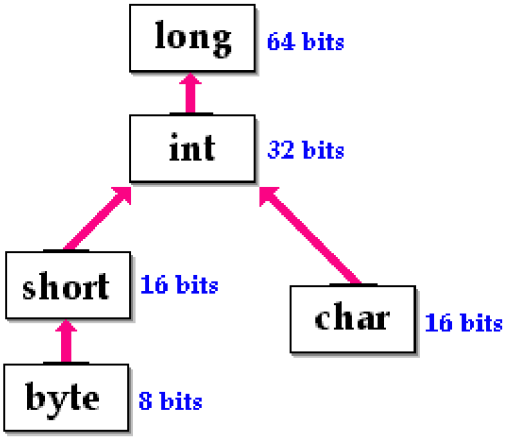
Dans tous les langages possédant la notion de sous-programme (ou fonction ou procédure), il se pose une question à savoir, les paramètres **formels** décrits lors de la déclaration d'un sous-programme ne sont que des variables muettes servant à expliquer le fonctionnement du sous-programme sur des futures variables lorsque le sous-programme s'exécutera **effectivement**.

La démarche en informatique est semblable à celle qui, en mathématiques, consiste à écrire la fonction f(x) = 3*x - 7, dans laquelle x alors une variable muette indiquant comment f est calculée : en informatique **elle joue le rôle du paramètre formel**. Lorsque l'on veut obtenir une valeur effective de la fonction mathématique f, par exemple pour x=2, on écrit f(2) et l'on calcule f(2)=3*2 - 7 = -1. En informatique on "**passera**" un paramètre effectif dont la valeur vaut 2 à la fonction. D'une manière générale, en informatique, il y a un sous-programme appelant et un sous-programme appelé par le sous-programme appelant.

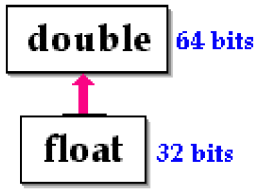
Compatibilité des types des paramètres

Resituons la compatibilité des types entier et réel en Java. Un moyen mnémotechnique pour retenir cette compatibilité est indiqué dans les figures ci-dessous, par la taille en nombre décroissant de bits de chaque type que l'on peut mémoriser sous la forme "**qui peut le plus peut le moins**" ou bien un type à **n** bits accueillera un sous-type à **p** bits, si **p** est inférieur à **n**.

Les types entiers compatibles :



Les types réels compatibles :



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

```
class Application5
{  public static void main(String[ ] args)
    { // recherche séquentielle dans un tableau
      int [ ] table= {12,-5,7,8,-6,6,4,78};
      byte elt = 4;
      short i ;
      for ( i = 0 ; i<8 ; i++ )
      if (elt= =table[i]) break ;
      afficher(i,elt);
    }
  static void afficher (int rang , long val)
  { if (rang == 8)
```

Appel de la méthode afficher

```
afficher(i,elt);
```

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

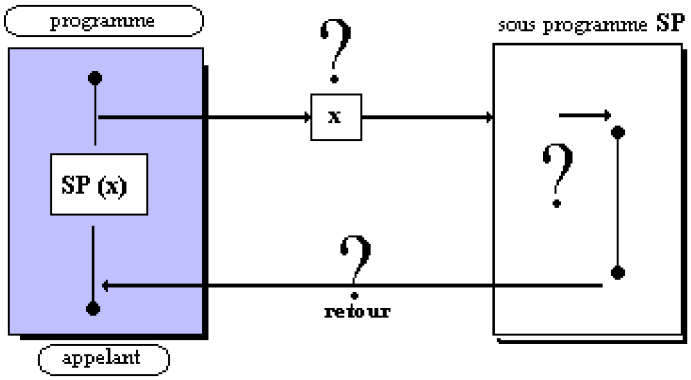
- Le paramètre effectif "**i**" est associé au paramètre formel **rang**.(**short** = entier signé sur 16 bits et **int** = entier signé sur 32 bits)

```
      System.out.println("valeur : "+val+" pas trouvée.");
    else
      System.out.println("valeur : "+val+" trouvée au rang :"+ rang);
  }
}
```

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

Les deux modes de transmission des paramètres

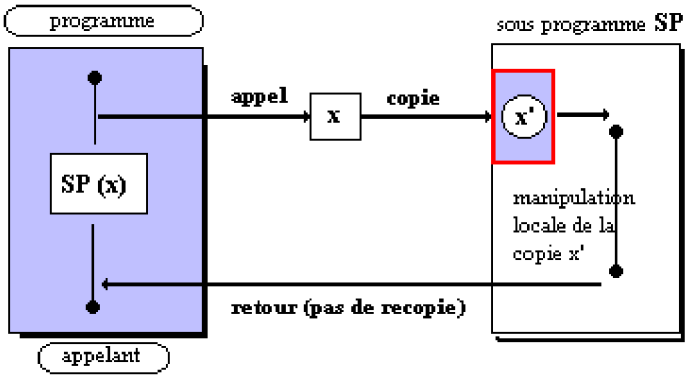
Un paramètre effectif transmis au sous-programme appelé est en fait un moyen d'utiliser ou d'accéder à une information appartenant au bloc appelant (le bloc appelé peut être le même que le bloc appelant, il s'agit alors de récursivité).



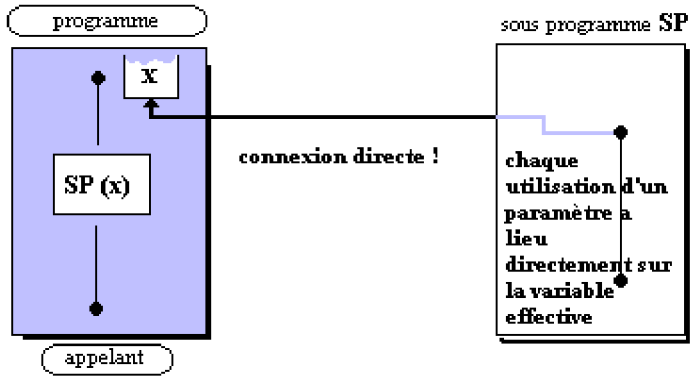
En Java, il existe deux modes de transmission (ou de passage) des paramètres (semblables à Delphi).

Le passage par **valeur** uniquement réservé à tous les types élémentaires

(**int**, **byte**, **short**, **long**, **boolean**, **double**, **float**, **char**).



Le passage par **référence** uniquement réservé à tous les types objets.



Remarque :

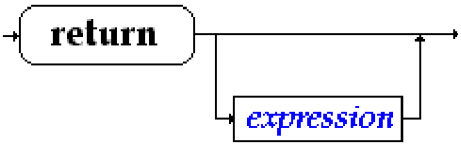
Le choix de **passage selon les types** élimine les inconvénients dûs à l'encombrement mémoire et à la lenteur de recopie de la valeur du paramètre par exemple dans un passage par valeur, car nous verrons plus loin que les **tableaux en Java sont des objets** et qu'ils sont donc passés **par référence**.

Les retours de résultat de méthode-fonction

Les méthodes en Java peuvent renvoyer un résultat de n'importe quel type élémentaire ou objet. Bien qu'une méthode ne puisse renvoyer qu'un seul résultat, l'utilisation du passage par référence d'objets permet aussi d'utiliser les paramètres de type objet d'une méthode comme des variables d'entrée/sortie.

En Java le retour de résultat est passé grâce au mot clef **return** placé n'importe où dans le corps de la méthode.

Syntaxe :



Sémantique :

- L'expression lorsqu'elle est présente est quelconque mais doit être obligatoirement du même type que le type du résultat déclaré dans l'en-tête de fonction (ou d'un type compatible).
- Lorsque le **return** est rencontré il y a arrêt de l'exécution du code de la méthode et retour du résultat dans le bloc appelant.
- Lorsqu'il n'y a pas d'expression après le **return**, le compilateur refusera cette éventualité sauf si vous êtes dans une méthode-procédure (donc du type void), le **return** fonctionne comme un **break** pour la méthode et interrompt son exécution.

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

```
class Application6
{  public static void main(String[ ] args)
    { // ...
      int x , y ;
      x = 4 ;
      y = f(5) ;
      y = f(x) ;
      System.out.println("f(x)="+ f(x) );
      System.out.println("f(5)="+ f(5) );
    }
  static int f (int x )
  {  return 3*x-7;
  }
}
```

Appel de la méthode f

f (5) ;
ou bien
f (x) ;

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

Exemple de méthode-procédure

```
class Application7
{  public static void main(String[ ] args)
    {
      int a = 0  ;
      procedure ( a );
      procedure ( a+4 );
    }
  static void procedure(int x)
   {
    if (x = = 0)
    { System.out.println("avant return");
       return ;
     }
    System.out.println("après return");
   }
}
```

Appel de la méthode procedure

Dans le cas du premier appel (x ==0) est true donc ce sont les instructions:
System.out.println("avant return");
return *;*
qui sont exécutées, puis interruption de la méthode.

Dans le cas du second appel (x ==0) est false c'est donc l'instruction:
System.out.println("avant return");
qui est exécutée, puis fin normale de la méthode.

4 - Visibilités des variables

Le principe de base est que les variables en Java sont visibles (donc utilisables) dans le bloc dans lequel elles ont été définies.

Visibilité de bloc

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

Toute variable déclarée dans un bloc est visible dans ce bloc et dans tous les blocs imbriqués dans ce bloc.

En java les blocs sont constitués par :

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

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

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

comme paramètre de la méthode,

comme variable locale à la méthode,

dans un bloc inclus dans la méthode.

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

Variables dans une classe, dans une méthode

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

Exemple de visibilité dans une classe

```
class ExempleVisible1 {
 int a = 10;

      int g (int x )
```

La variable "a" définie dans **int a** =10; :

- Est une variable de la classe ExempleVisible.

```
    { return  3*x-a;
    }

    int f (int x, int a )
    { return 3*x-a;
    }

  }
```

- Elle est visible dans la méthode **g** et dans la méthode **f**. C'est elle qui est utilisée dans la méthode **g** pour évaluer l'expression 3*x-**a.**

- Dans la méthode **f**, elle est masquée par le paramètre du même nom qui est utilisé pour évaluer l'expression 3*x-**a.**

Contrairement à ce que nous avions signalé plus haut nous n'avons pas présenté un exemple fonctionnant sur des méthodes de classes (qui doivent obligatoirement être précédées du mot clef **static**), mais sur des méthodes d'instances dont nous verrons le sens plus loin en POO.

Remarquons avant de présenter le même exemple cette fois-ci sur des méthodes de classes, que quelque soit le genre de méthode la visibilité des variables est **identique**.

Exemple identique sur des méthodes de classe

```
class ExempleVisible2 {
 static int a = 10;

    static int g (int x )
    { return  3*x-a;
    }

    static int f (int x, int a )
    { return 3*x-a;
    }

  }
```

La variable "a" définie dans **static int a** =10; :

- Est une variable de la classe ExempleVisible.

- Elle est visible dans la méthode **g** et dans la méthode **f**. C'est elle qui est utilisée dans la méthode **g** pour évaluer l'expression 3*x-**a.**

- Dans la méthode **f**, elle est masquée par le paramètre du même nom qui est utilisé pour évaluer l'expression 3*x-**a.**

Les variables définies dans une méthode (de classe ou d'instance) subissent les règles classiques de la visibilité du bloc dans lequel elles sont définies :

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

Reprenons l'exemple précédent en adjoignant des variables locales aux deux méthodes f et g.

Exemple de variables locales

```
class ExempleVisible3 {
 static int a = 10;

    static int g (int x )
```

La variable de classe "a" définie dans **static int a** = 10; est masquée dans les deux méthodes **f** et **g**.

```
    {  char car = 't';
       long a = 123456;
        ....
       return  3*x-a;
    }

    static int f (int x, int a )
    {  char car ='u';
        ....
       return 3*x-a;
    }

}
```

Dans la méthode **g**, c'est la variable locale **long**a = 123456 qui masque la variable de classe **static int a**. **char** car ='t'; est une variable locale à la méthode **g**.

- Dans la méthode **f**, **char** car ='u'; est une variable locale à la méthode **f**, le paramètre **int**a masque la variable de classe **static int a**.

Les variables locales **char** car n'existent que dans la méthode où elles sont définies, les variables "car" de **f** et celle de **g** n'ont aucun rapport entre elles, bien que portant le même nom.

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

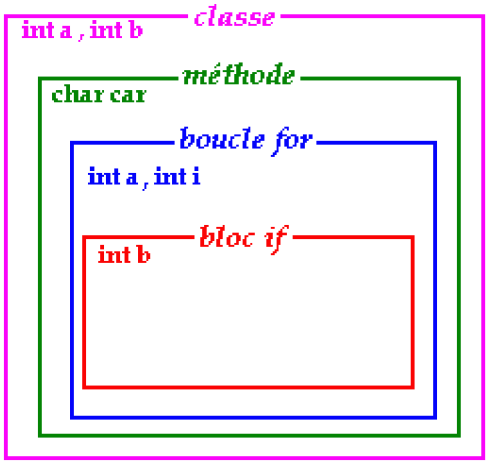
Les variables définies dans des blocs du genre instructions composées, boucles, try..catch ne sont visibles que dans le bloc et ses sous-blocs imbriqués, dans lequel elle sont définies.

Toutefois attention aux ***redéfinitions*** de variables locales. Les blocs du genre instructions composées, boucles, try..catch ne sont utilisés qu'à l'intérieur du corps d'une méthode (ce sont les actions qui dirigent le fonctionnement de la méthode), les variables définies dans de tels blocs sont automatiquement considérées par Java comme des variables locales à la méthode. Tout en respectant à l'intérieur d'une méthode le principe de visibilité de bloc, Java **n**'accepte **pas** le masquage de variable à l'**intérieur des blocs imbriqués**.

Nous donnons des exemples de cette visibilité :

Exemple correct de variables locales

```
class ExempleVisible4 {

    static int a = 10, b = 2;

     static int f (int x )

     {  char car = 't';

        for (int i = 0;  i < 5 ;  i++)
        {int a=7;

          if (a < 7)
           {int b = 8;
            b = 5-a+i*b;
```



```
        }

        else b = 5-a+i*b;
        }

        return  3*x-a+b;
    }
}
```

La variable de classe "a" définie dans **static int a** = 10; est masquée dans la méthode **f** dans le bloc imbriqué **for**.

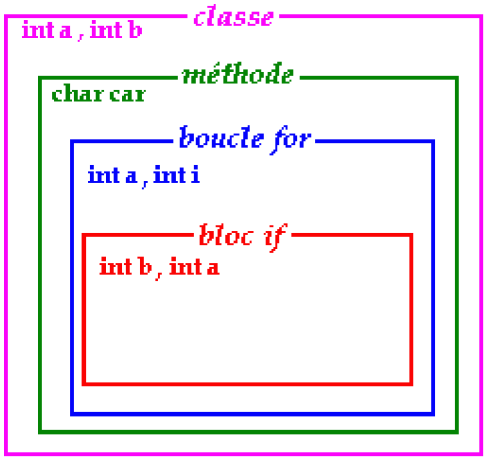
La variable de classe "b" définie dans **static int b** = 2; est masquée dans la méthode **f** dans le bloc imbriqué **if**.

Dans l'instruction **{ intb** = 8; **b** = 5-**a**+**i***b; **}** , c'est la variable **b** interne à ce bloc qui est utilisée car elle masque la variable **b** de la classe.

Dans l'instruction **else b** = 5-**a**+**i***b; , c'est la variable **b** de la classe qui est utilisée (car la variable **int b** = 8 n'est plus visible ici) .

Exemple de variables locales générant une erreur

```
class ExempleVisible5 {

    static int a = 10, b = 2;

     static int f (int x )

     {  char car = 't';

        for (int i = 0;  i < 5 ;  i++)
        {int a=7;

          if (a < 7)
          {int b = 8, a = 9;
           b = 5-a+i*b;
          }

        else b = 5-a+i*b;
        }

        return  3*x-a+b;
    }
}
```



Toutes les remarques précédentes restent valides puisque l'exemple ci-contre est quasiment identique au précédent. Nous avons seulement rajouté dans le bloc **if** la définition d'une nouvelle variable interne **a** à ce bloc.

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

Remarquons que le principe de visibilité des variables adopté en Java est identique au principe inclus dans tous les langages à structures de bloc y compris pour le **masquage**, s'y rajoute en Java uniquement l'interdiction de la **redéfinition** à l'intérieur d'une même méthode (semblable en fait, à l'interdiction de redéclaration sous le même nom, de variables locales à un bloc).

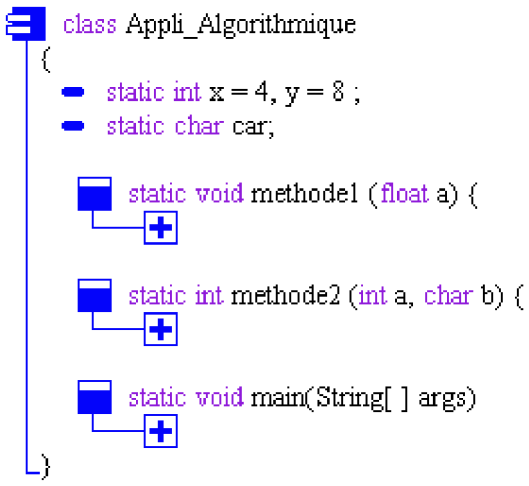
Synthèse : utiliser une méthode static dans une classe

❖ Les méthodes représentent les actions

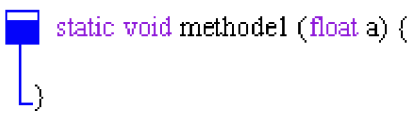
En POO, les méthodes d'une classe servent à indiquer comment fonctionne un objet dans son environnement d'exécution. Dans le cas où l'on se restreint à utiliser Java comme un langage algorithmique, la classe représentant le programme principal, les méthodes représenteront les sous programmes du programme principal. C'est en ce sens qu'est respecté le principe de la programmation structurée.

Attention, il est impossible en Java de déclarer une méthode à l'intérieur d'une autre méthode comme en pascal; toutes les méthodes sont au même niveau de déclaration : ce sont les méthodes de la classe !

Typiquement une application purement algorithmique en Java aura donc cette forme :
(un programme principal "main" et ici, deux sous-programmes methode1 et methode2)

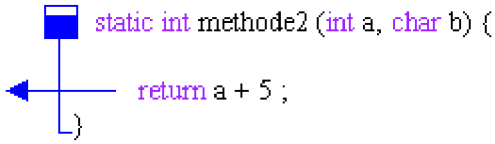


Les méthodes type procédure (méthode pouvant avoir plusieurs paramètres en entrée, mais ne renvoyant pas de résultat) sont précédées du mot clef **void** :



Les autres méthodes (précédées d'un mot clef typé comme : **int**, **char**, **byte**, **boolean**,...) sont des fonctions pouvant avoir plusieurs paramètres en entrée qui renvoient obligatoirement un

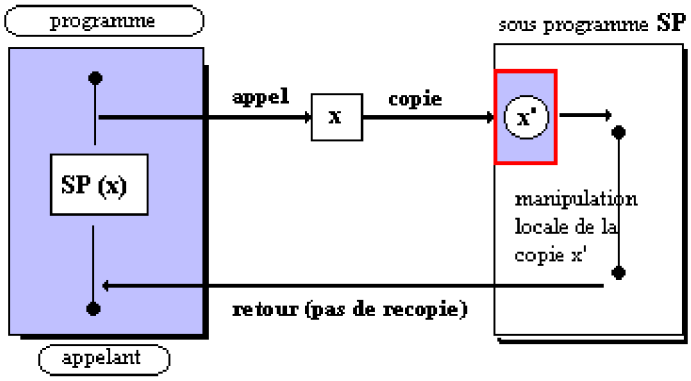
résultat du type déclaré par le mot clef précédant le nom de la fonction :



La méthode précédente possède 2 paramètres en entrée **int** a et **char** b, et renvoie un paramètre de type **int**.

❖ Les paramètres Java sont passés par valeur

Le passage des paramètres par valeur entre un programme appelant (**main** ou tout autre méthode) et un sous programme (les méthodes) s'effectue selon le schéma ci-dessous :



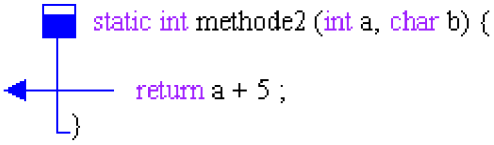
En Java tous les paramètres sont passés par valeur (même si l'on dit que **Java passe les objets par référence**, en fait il s'agit très précisément de **la référence de l'objet qui est passée par valeur**). Pour ce qui est de la vision algorithmique de Java, le passage par valeur est la norme.

Ainsi aucune variable ne peut être passée comme paramètre à la fois d'entrée et de sortie comme en Pascal.

Comment faire en Java si l'on souhaite malgré tout passer une variable à la fois en entrée et en sortie ?

Il faut la passer comme paramètre effectif (passage par valeur), et lui réaffecter le résultat de la fonction. L'exemple suivant indique comment procéder :

soit la méthode précédente recevant un **int** et un **char** et renvoyant un **int**



Les instructions ci-après permettent de modifier la valeur d'une variable x à l'aide de la méthode "methode2" :

```
int x = 10;
   // avant l'appel x vaut 10
x = methode2 ( x , '@');
  // après l'appel x vaut 15
```

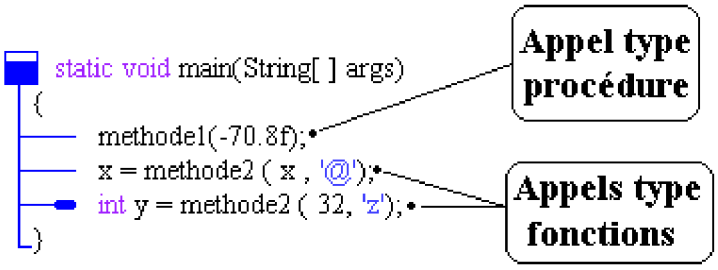
❖ Appeler une méthode en Java

1. Soit on appelle la méthode comme une procédure en Pascal (avec ses paramètres effectifs éventuels), soit on l'appelle comme une fonction en utilisant son résultat. Reprenons les deux méthodes précédentes et appelons les :

```
static void methode1 (float a) {
}
```

```
static int methode2 (int a, char b) {
    return a + 5 ;
}
```

2. Soit on peut appeler ces deux méthodes dans la méthode "main" :

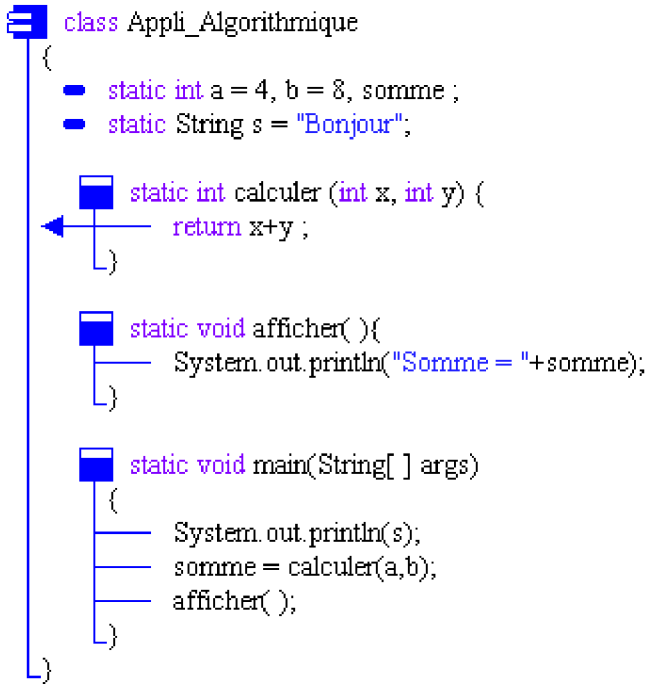


3. Soit on peut les appeler dans une autre méthode "methode3" :

```
static boolean methode3 (int a) {
    methode1(-70.8f);
    x = methode2 ( x , '@');
    int y = methode2 ( 32, 'z');
    return true ;
}
```

❖ Exemple de programme : calcul d'une somme

avec méthodes *procédure* et *fonction*



```
class Appli_Algorithmique
{
   static int a = 4, b = 8, somme ;
   static String s = "Bonjour";

   static int calculer (int x, int y) {
      return x+y ;
   }

   static void afficher( ){
      System.out.println("Somme = "+somme);
   }

   static void main(String[ ] args)
   {
      System.out.println(s);
      somme = calculer(a,b);
      afficher( );
   }
}
```

Résultats d'exécution de ce programme :

Bonjour
Somme = 12

Structures de données de base

Les chaînes String

Java2

La classe String

Le type de données **String** (chaîne de caractère) n'est pas un type élémentaire en Java, c'est une classe. Donc une chaîne de type **String** est un objet qui n'est utilisable qu'à travers les méthodes de la classe **String**.

Pour accéder à la classe String et à toutes ses méthodes, vous devez mettre avant la déclaration de votre classe l'instruction d'importation de package suivante :

import java.lang.String ;

Un littéral de chaîne est une suite de caractères entre guillemets : " abcdef " est un exemple de littéral de String.

Etant donné que cette classe est très utilisée les variables de type **String** bénéficient d'un statut d'utilisation aussi souple que celui des autres types élémentaires. On peut les considérer comme des listes de caractères numérotés de 0 à n-1 (si n figure le nombre de caractères de la chaîne).

Déclaration d'une variable String	**String** str1;
Déclaration d'une variable String avec initialisation	**String** str1 = " abcdef "; Ou **String** str1 = **new String**("abcdef ");
On accède à la longueur d'une chaîne par la méthode : **int** length( )	**String** str1 = "abcdef"; **int** longueur; longueur = str1.length( ); *// ici longueur vaut 5*

Toutefois les **String** de Java sont moins conviviales en utilisation que les **string** de pascal ou celles de C#, il appartient au programmeur d'écrire lui-même ses méthodes d'**insertion**, **modification** et **suppression**.

Toutes les autres manipulations sur des objets **String** nécessitent l'emploi de méthodes de la classe String. Nous donnons quelques exemples d'utilisation de méthode classique sur les **String**.

Le type **String** possède des méthodes classiques d'extraction, de concaténation, de changement de casse, etc.

<table>
<tr>
<td>Concaténation de deux chaînes

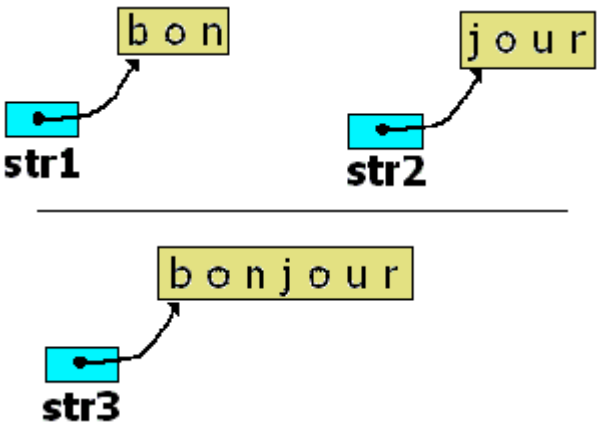
Un opérateur ou une méthode

Opérateur : + sur les chaînes

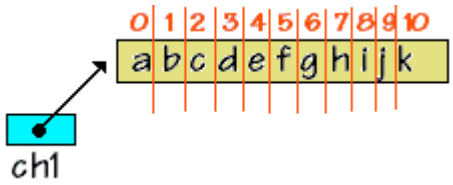
ou

Méthode : String concat(String s)

Les deux écritures ci-dessous sont donc équivalentes en Java :

str3 = str1+str2 ⇔ str3 = str1.concat(str2)</td>
<td>String str1,str2,str3;
str1="bon";
str2="jour";
str3=str1+str2;
 
</td>
</tr>
<tr>
<td>On accède à un caractère de rang fixé d'une chaîne par la méthode :

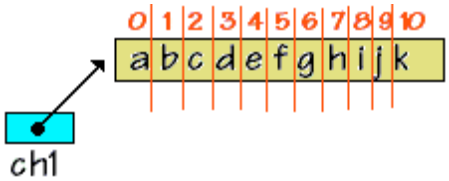
char charAt(int rang)

Il est possible d'accéder en lecture seulement à chaque caractère d'une chaîne, mais qu'il est impossible de modifier un caractère directement dans une chaîne.</td>
<td>String ch1 = "abcdefghijk";
 

char car = ch1.charAt(4);

// ici la variable car contient la lettre 'e'</td>
</tr>
<tr>
<td>position d'une sous-chaîne à l'intérieur d'une chaîne donnée :

méthode :

int indexOf (String ssch)</td>
<td>String ch1 = " abcdef " , ssch="cde";
 

int rang ;
rang = ch1.indexOf (ssch);

// ici la variable rang vaut 2</td>
</tr>
</table>

Les **String** Java ne peuvent pas être considérées comme des tableaux de caractères, il est nécessaire, si l'on souhaite se servir d'une **String**, d'utiliser la méthode **toCharArray** pour convertir la chaîne en un tableau de caractères contenant tous les caractères de la chaîne.

Enfin, attention ces méthodes de manipulation d'une chaîne ne modifient pas la chaîne objet qui invoque la méthode mais renvoient un autre objet de chaîne différent. Ce nouvel objet est obtenu après action de la méthode sur l'objet initial.

Soient les quatre lignes de programme suivantes :

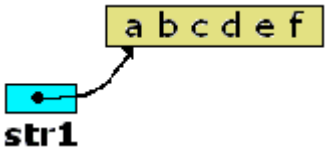
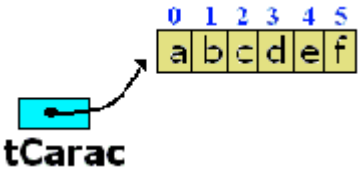
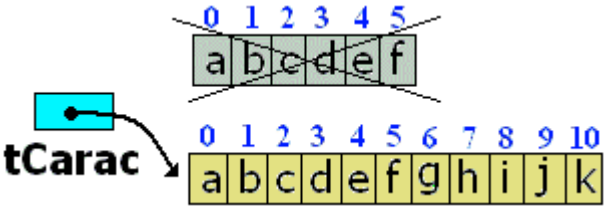
String str1 = "abcdef" ;

char [ ] tCarac ;

tCarac = str1.**toCharArray**( ) ;

tCarac = "abcdefghijk".**toCharArray**( );

Illustrons ci-dessous ce qui se passe relativement aux objets créés :

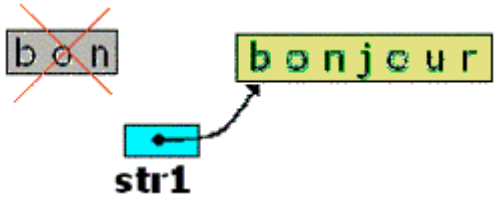
String str1 = "abcdef" ;	 str1 référence un objet de chaîne.
char [ ] tCarac ; tCarac = str1.**toCharArray**( ) ;	 tCarac référence un objet de tableau à 6 éléments.
tCarac = "abcdefghijk".**toCharArray**( );	 tCarac référence maintenant un nouvel objet de tableau à 11 éléments, l'objet précédent est perdu (éligible au Garbage collector)

L'exemple précédent sur la concaténation ne permet pas de voir que l'opérateur + ou la méthode concat renvoie réellement un nouvel objet en particulier lors de l'écriture des quatre lignes

suivantes :

String str1,str2;
str1="bon";
str2="jour";
str1=str1+str2;

Illustrons ici aussi ce qui se passe relativement aux objets créés :

String str1,str2; str1="bon"; str2="jour";	
str1=str1+str2;	 un nouvel objet de chaîne a été créé et str1 "pointe" maintenant vers lui.

Opérateurs d'égalité de String

- L'opérateur d'égalité = = , détermine si deux objets **String** spécifiés ont la **même référence et non la même valeur**, il ne se comporte pas en Java comme sur des éléments de type de base (int, char,...)

  String a , b ;
  (a = = b) renvoie **true** si les variables a et b référencent chacune le même objet de chaîne sinon il renvoie **false**.

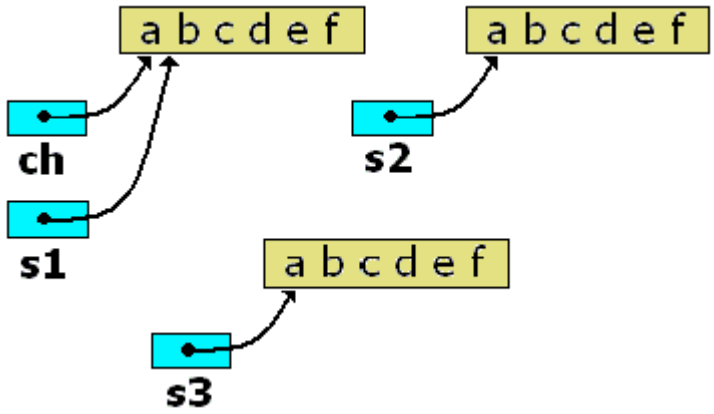
- La méthode **boolean** equals(Object s) teste si deux chaînes n'ayant pas la même référence ont la même valeur.

  String a , b ;
  a.equals (b) renvoie **true** si les variables a et b ont la même valeur sinon il renvoie **false**.

En d'autres termes si nous avons deux variables de String ch1 et ch2, que nous ayons écrit ch1 = "abcdef"; et plus loin ch2 = "abcdef"; les variables ch1 et ch2 n'ont pas la même référence mais ont la même valeur (valeur = "abcdef").

Voici un morceau de programme qui permet de tester l'opérateur d'égalité = = et la méthode equals :

```
String s1,s2,s3,ch;
ch = "abcdef";
s1 = ch;
s2 = "abcdef";
s3 = new String("abcdef".toCharArray( ));
```




```
System.out.println("s1="+s1);
System.out.println ("s2="+s2);
System.out.println ("s3="+s3);
System.out.println ("ch="+ch);
if( s1 == ch ) System.out.println ("s1=ch");
  else System.out.println ("s1<>ch");
if( s1 == s3 ) System.out.println ("s1=s3");
  else System.out.println ("s1<>s3");

if( s1.equals(s2) ) System.out.println ("s1 même val. que s2");
  else System.out.println ("s1 différent de s2");

if( s1.equals(s3) ) System.out.println ("s1 même val. que s3");
  else System.out.println ("s1 différent de s3");

if( s1.equals(ch) ) System.out.println ("s1 même val. que ch");
  else System.out.println ("s1 différent de ch");
```

Après exécution on obtient :

```
s1=abcdef
s2=abcdef
s3=abcdef
ch=abcdef
s1=ch
s1<>s3
s1 même val. que s2
s1 même val. que s3
s1 même val. que ch
```

ATTENTION

En fait, Java a un problème de cohérence avec les littéraux de String. Le morceau de programme ci-dessous montre cinq évaluations équivalentes de la String s2 qui contient après l'affectation la chaîne "abcdef", puis deux tests d'égalité utilisant l'opérateur = = . Nous avons mis en commentaire, après chacune des cinq affectations, le résultat des deux tests :

```
String ch;
ch = "abcdef" ;

String s2,s4="abc" ;
s2 = s4.concat("def") ; /* après tests : s2<>abcdef, s2<>ch */
s2 = "abc".concat("def"); /* après tests : s2<>abcdef, s2<>ch */
s2 = s4+"def"; /* après tests : s2<>abcdef, s2<>ch */

s2="abc"+"def"; /* après tests : s2 ==abcdef, s2 == ch */
s2="abcdef"; /* après tests : s2 == abcdef, s2 == ch */

//-- tests d'égalité avec l'opérateur = =
if( s2 == "abcdef" ) System.out.println ("s2==abcdef");
else System.out.println ("s2<>abcdef");
if( s2 == ch ) System.out.println ("s2==ch");
else System.out.println ("s2<>ch");
```

Nous remarquons que selon que l'on utilise ou non des littéraux les résultats du test ne sont pas les mêmes.

CONSEIL

Pour éviter des confusions et mémoriser des cas particuliers, il est conseillé d'utiliser la méthode **equals** pour tester la valeur d'égalité de deux chaînes.

Rapport entre String et char

Une chaîne **String** contient des éléments de base de type **char** comment passe-t-on de l'un à l'autre type.

1°) On ne peut pas considérer un **char** comme un cas particulier de **String**, le transtypage suivant est refusé :

```
char car = 'r';
String s;
s = (String)car;
```

Il faut utiliser la méthode de conversion valueOf des **String** :

```
s = String.valueOf(car);
```

2°) On peut concaténer avec l'opérateur +, des **char** à une chaîne **String** déjà existante et affecter le résultat à une String :

```
String s1 , s2 ="abc" ;
char c = 'e' ;
s1 = s2 + 'd' ;
s1 = s2 + c ;
```

L'écriture suivante sera refusée :	Ecriture correcte associée :
String s1 , s2 ="abc" ; **char** c = 'e' ; s1 = 'd' + c ; *// types incompatibles* s1 = 'd' + 'e'; *// types incompatibles*	**String** s1 , s2 ="abc" ; **char** c = 'e' ; s1 = "d" + **String**.valueOf (c) ; s1 = "d" + "e";

- Le caractère 'e' est de type **char**,
- La chaîne "e" est de type **String** (elle ne contient qu'un seul caractère)

Pour plus d'information sur toutes les méthode de la classe String voici telle qu'elle est présentée par Sun dans la documentation du JDK 1.4.2 (**http://java.sun.com**), la liste des méthodes de cette classe.

Tableaux et matrices

Java2

Dès que l'on travaille avec de nombreuses données homogènes (de même type) la première structure de base permettant le regroupement de ces données est le **tableau.** Java comme tous les langages algorithmiques propose cette structure au programmeur. Comme pour les **String**, pour des raisons d'efficacité dans l'encombrement mémoire, les tableaux sont gérés par Java, comme des objets.

- Les tableaux Java sont comme en Delphi, des tableaux de tous types y compris des types objets.
- Il n'y a pas de mot clef spécifique pour la classe tableaux, mais l'opérateur symbolique **[ ]** indique qu'une variable de type fixé est un tableau.
- La taille d'un tableau doit obligatoirement avoir été définie avant que Java accepte que vous l'utilisiez !

Les tableaux à une dimension

Déclaration d'une variable de tableau :
int [ ] table1;
char [ ] table2;
float [ ] table3;
...
String [ ] tableStr;

Déclaration d'une variable de tableau avec définition explicite de taille :
int [ ] table1 = **new int** [5];
char [] table2 = **new char** [12];
float [ ] table3 = **new float** [8];
...
String [ ] tableStr = **new String** [9];

Le mot clef **new** correspond à la **création d'un nouvel objet** (un nouveau tableau) dont la taille est fixée par la valeur indiquée entre les crochets. Ici 4 tableaux sont créés et prêts à être utilisés : table1 contiendra 5 entiers 32 bits, table2 contiendra 12 caractères, table3 contiendra 8 réels en simple précision et tableStr contiendra 9 chaînes de type **String**.

On peut aussi déclarer un tableau sous la forme de deux instructions : une instruction de déclaration et une instruction de définition de taille avec le mot clef **new**, la seconde pouvant être mise n'importe où dans le corps d'instruction, mais elle doit être utilisée avant toute manipulation du tableau. Cette dernière instruction de définition peut être répétée plusieurs fois dans le programme, il s'agira alors à chaque fois de la **création d'un nouvel objet** (donc un nouveau tableau), l'**ancien étant détruit** et désalloué automatiquement par le ramasse-miettes (garbage collector) de Java.

```
int [ ] table1;
char [ ] table2;
float [ ] table3;
String [ ] tableStr;
....
table1 = new int [5];
table2  = new char [12];
table3  = new float [8];
tableStr  = new String [9];
```

Déclaration et initialisation d'un tableau avec définition implicite de taille :

```
int [ ] table1 = {17,-9,4,3,57};
char [ ] table2  = {'a','j','k','m','z'};
float [ ] table3  = {-15.7f,75,-22.03f,3,57};
String [ ] tableStr  = {"chat","chien","souris","rat","vache"};
```

Dans cette éventualité Java crée le tableau, calcule sa taille et l'initialise avec les valeurs fournies.

Il existe un attribut général de la classe des tableaux, qui contient **la taill**e d'un tableau quelque soit son type, c'est l'attribut length.

Exemple :

```
int [ ] table1 = {17,-9,4,3,57};
int taille;
taille = table1.length; // taille = 5
```

Il existe des classes permettant de manipuler les tableaux :

- La classe **Array** dans le package **java.lang.reflect.Array**, qui offre des méthodes de classe permettant de créer dynamiquement et d'accéder dynamiquement à des tableaux.

- La classe **Arrays** dans le package **java.util.Arrays**, offre des méthodes de classe pour la recherche et le **tri** d'éléments d'un tableau.

Utiliser un tableau

Un tableau en Java comme dans les autres langages algorithmiques, s'utilise à travers une cellule de ce tableau repérée par un indice obligatoirement de type entier ou un char considéré comme un entier (byte, short, int, long ou char).

Le premier élément d'un tableau est numéroté **0**, le dernier **length-1**.

On peut ranger des valeurs ou des expressions du type général du tableau dans une cellule du tableau.

Exemple avec un tableau de type **int** :

```
int [ ] table1 = new int [5];
// dans une instruction d'affectation:
table1[0] = -458;
table1[4] = 5891;
table1[5] = 72;  <---  erreur de dépassement de la taille ! (valeur entre 0 et 4)

    // dans une instruction de boucle:
    for (int i = 0 ; i<= table1.length-1;  i++)
     table1[i] = 3*i-1; // après la boucle: table1 = {-1,2,5,8,11}
```

Même exemple avec un tableau de type **char** :

```
char [] table2  = new char [7];

    table2[0] = '?' ;
    table2[4] = 'a' ;
    table2[14] = '#' ;  <--- est une erreur de dépassement de la taille
    for (int i = 0 ; i<= table2.length-1;  i++)
         table2[i] =(char)('a'+i);


    //-- après la boucle: table2 = {'a', 'b', 'c' ,'d', 'e', 'f'}
```

Remarque :

Dans une classe exécutable la méthode main reçoit en paramètre un tableau de String nommé args qui correspond en fait aux éventuels paramètres de l'application elle-même:

public static void main(String [ ] args)

Les tableaux à deux dimension : matrices

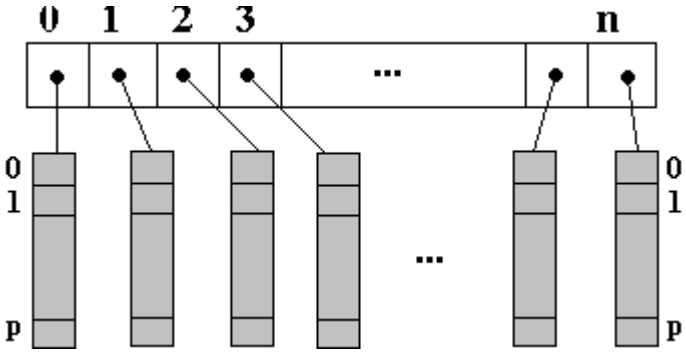
Les tableaux en Java peuvent être à deux dimensions (voir même à plus de deux) auquel cas on peut les appeler des matrices, ce sont aussi des objets et ils se comportent comme les tableaux à une dimension tant au niveau des déclarations qu'au niveau des utilisations. La déclaration s'effectue avec deux opérateurs crochets **[ ] [ ]** . Les matrices Java ne sont pas en réalité des vraies matrices, elles ne sont qu'un cas particulier des tableaux multi indices.

Leur structuration n'est pas semblable à celle des tableaux pascal, en fait en java une matrice est composée de plusieurs tableaux unidimensionnels de même taille (pour fixer les idées nous les appellerons les lignes de la matrice) : dans une déclaration Java le premier crochet sert à indiquer le nombre de lignes (nombre de tableaux à une dimension), le second crochet sert à indiquer la taille de la ligne.

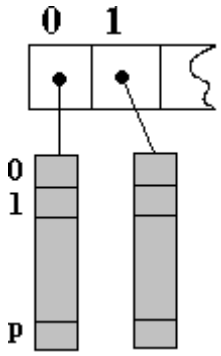
Un tel tableau à deux dimensions, peut être considéré comme un tableau unidimensionnel de pointeurs, où chaque pointeur référence un autre tableau unidimensionnel.

Voici une manière imagée de visualiser une matrice à n+1 lignes et à p+1 colonnes

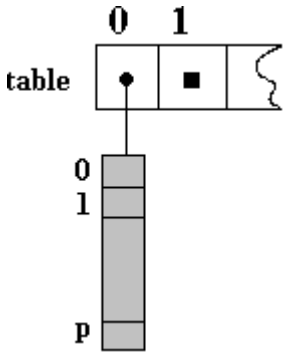
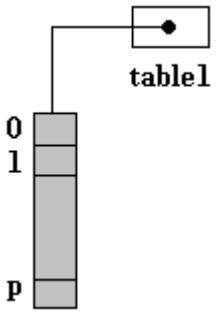
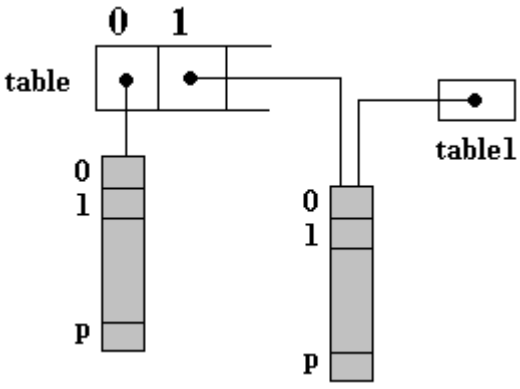
int [ ][ ] table = **new int** [n+1][p+1];



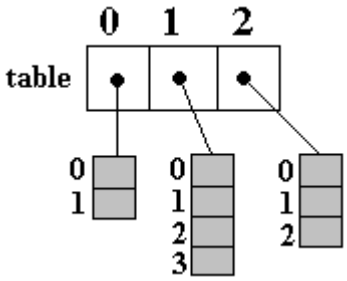
Les tableaux multiples en Java sont utilisables comme des tableaux unidimensionnels. Si l'on garde bien présent à l'esprit le fait qu'une cellule contient une référence vers un autre tableau, on peut alors écrire en Java soient des instructions pascal like comme table[i,j] traduite en java par table[i][j], soient des instructions spécifiques à java n'ayant pas d'équivalent en pascal comme dans l'exemple ci-après :

table[0] = **new** int[p+1];
 	
table[1] = **new** int[p+1];
 	
Dans chacune de ces deux instructions nous créons un objet de tableau unidimensionnel qui est référencé par la cellule de rang 0, puis par celle de rang 1.
	

Ou encore, en illustrant ce qui se passe après chaque instruction :

int [ ][ ] table = **new int** [n+1][p+1]; table[0] = **new** int[p+1];	
int [ ] table1 = **new int** [p+1];	
table[1] = table1 **;**	

Rien n'oblige les tableaux référencés d'avoir la même dimension, ce type de tableau se dénomme tableaux en escalier ou tableaux déchiquetés en Java :

int [ ][ ] table = **new int** [3][ ];
table[0] = **new** int[2];
table[1] = **new** int[4];
table[2] = **new** int[3];
	

Si l'on souhaite réellement utiliser des matrices (dans lequel toutes les lignes ont la même dimension) on emploiera l'écriture pascal-like, comme dans l'exemple qui suit.

Exemple d'écritures conseillées de matrice de type **int** :

```
int [ ][ ] table1 = new int [2][3];// deux lignes de dimension 3 chacune
// dans une instruction d'affectation:
table1[0][0] = -458;
table1[2][5] = -3;  <--- est une erreur de dépassement ! (valeur entre 0 et 1)
table1[1][4] = 83;  <--- est une erreur de dépassement ! (valeur entre 0 et 4)

    // dans une instruction de boucle:
    for (int i = 0 ; i<= 2;  i++)
     table1[1][i] = 3*i-1;

    // dans une instruction de boucles imbriquées:
    for (int i = 0 ; i<= 2;  i++)
     for (int k= 0 ; i<= 3;  i++) table1[i][k] = 100;
```

Information sur la taille d'un tableau multi-indices :

Le même attribut général length de la classe des tableaux, contient **la taill**e du tableau :

Exemple : matrice à deux lignes et à 3 colonnes

```
int [ ][ ] table1 = new int [2][3];
int taille;


taille = table1.length; // taille = 2 (nombre de lignes)

taille = table1[0].length; // taille = 3 (nombre de colonnes)

taille = table1[1].length; // taille = 3 (nombre de colonnes)
```

Java initialise les tableaux par défaut à 0 pour les **int**, **byte**, ... et à **null** pour les objets.

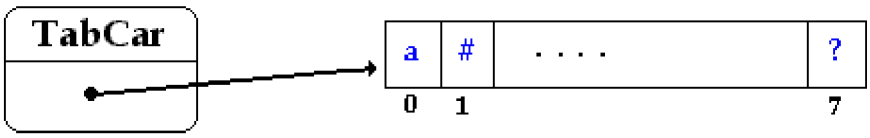
Tableaux dynamiques et listes

Java2

Tableau dynamique

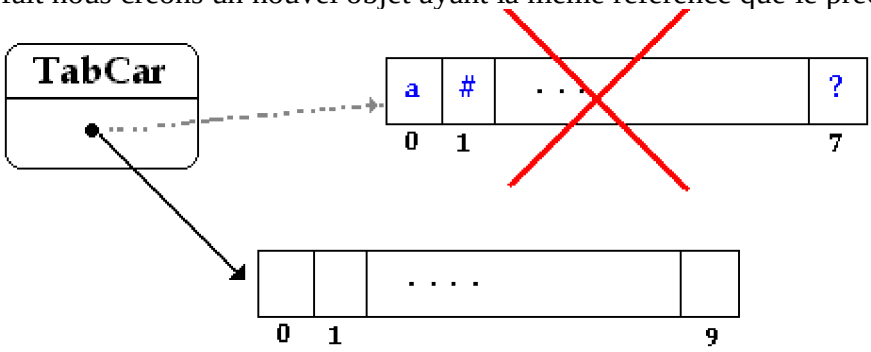
Un tableau array à une dimension, lorsque sa taille a été fixée soit par une définition explicite, soit par une définition implicite, **ne peut plus changer de taille**, c'est donc une structure statique.

char [ ] TableCar ;
TableCar = **new char**[8]; //définition de la taille et création d'un nouvel objet tableau à 8 cellules
TableCar[0] = 'a';
TableCar[1] = '#';
...
TableCar[7] = '?';



Si l'on rajoute l'instruction suivante aux précédentes

<TableCar= **new char**[10]; > il y a création d'un nouveau tableau de même nom et de taille 10, l'ancien tableau à 8 cellules est alors détruit. Nous ne redimensionnons pas le tableau, mais en fait nous créons un nouvel objet ayant la même reférence que le précédent :



Ce qui nous donne après exécution de la liste des instructions ci-dessous, un tableau TabCar ne contenant plus rien :
char [] TableCar ;
TableCar = **new char**[8];
TableCar[0] = 'a';
TableCar[1] = '#';
...
TableCar[7] = '?';
TableCar= **new char**[10];

Si l'on veut "agrandir" un tableau pendant l'exécution il faut en déclarer un nouveau plus grand et

recopier l'ancien dans le nouveau.

Il est possible d'éviter cette façon de faire en utilisant un vecteur (tableau unidimensionnel dynamique) de la classe Vector, présent dans le package **java.util.Vector**. Ce sont en fait des listes dynamiques gérées comme des tableaux.

Un objet de classe **Vector** peut "grandir" automatiquement d'un certain nombre de cellules pendant l'exécution, c'est le programmeur qui peut fixer la valeur d'augmentation du nombre de cellules supplémentaires dès que la capacité maximale en cours est dépassée. Dans le cas où la valeur d'augmentation n'est pas fixée, c'est la machine virtuelle Java qui procède à une augmentation par défaut (doublement dès que le maximum est atteint).

Vous pouvez utiliser le type **Vector** uniquement dans le cas d'objets et non d'éléments de type élémentaires (byte, short, int, long ou char ne sont pas autorisés), comme par exemple les String ou tout autre objet de Java ou que vous créez vous-même.

Les principales méthodes permettant de manipuler les éléments d'un Vector sont :

void addElement(**Object** obj)	ajouter un élément à la fin du vecteur
void clear( )	effacer tous les éléments du vecteur
Object elementAt(**int** index)	élément situé au rang = 'index'
int indexOf(**Object** elem)	rang de l'élément 'elem'
Object remove(**int** index)	efface l'élément situé au rang = 'index'
void setElementAt(**Object** obj, **int** index)	remplace l'élément de rang 'index' par obj
int size( )	nombre d'éléments du vecteur

Voici un exemple simple de vecteur de chaînes utilisant quelques unes des méthodes précédentes :

```
static void afficheVector(Vector vect)
//affiche un vecteur de String
 {
   System.out.println( "Vector taille = " + vect.size( ) );
   for ( int i = 0; i<= vect.size( )-1; i++ )
       System.out.println( "Vector[" + i + "]=" + (String)vect.elementAt( i ) );
 }

static void VectorInitialiser( )
// initialisation du vecteur de String
{    Vector table = new Vector( );
     String str = "val:";
     for ( int i = 0; i<=5; i++ )
          table.addElement(str + String.valueOf( i ) );
     afficheVector(table);
}
```

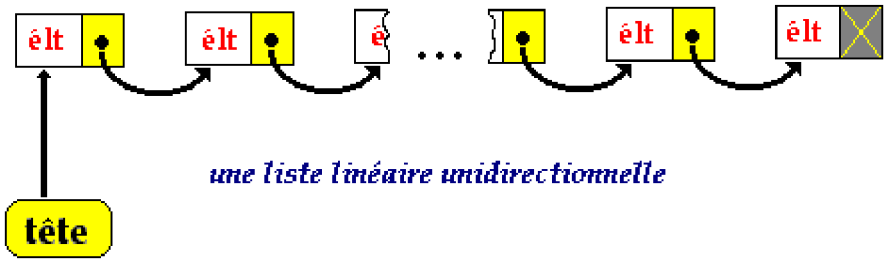
*Voici le résultat de l'exécution de la méthode***VectorInitialiser** ***:***

```
Vector taille = 6
Vector[0] = val:0
Vector[1] = val:1
Vector[2] = val:2
Vector[3] = val:3
Vector[4] = val:4
Vector[5] = val:5
```

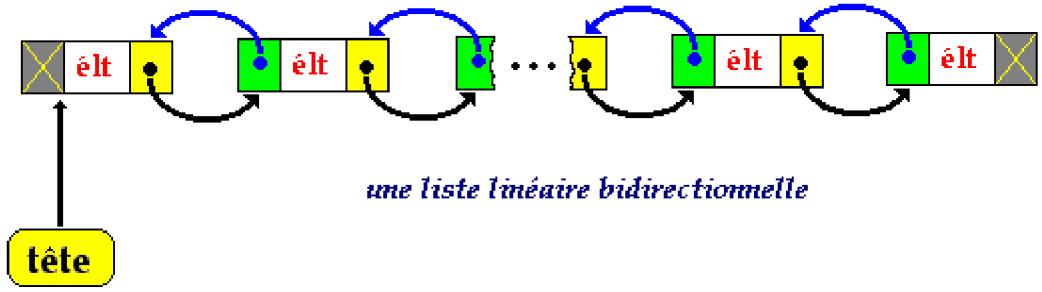
Les listes chaînées

Rappelons qu'une liste linéaire (ou liste chaînée) est un ensemble ordonné d'éléments de même type (structure de donnée homogène) auxquels on accède séquentiellement. Les opérations minimales effectuées sur une liste chaînée sont l'insertion, la modification et la suppression d'un élément quelconque de la liste.

Les listes peuvent être uni-directionnelles, elles sont alors parcourues séquentiellement dans un seul sens :



ou bien bi-directionnelles dans lesquelles chaque élément possède deux liens de chaînage, l'un sur l'élément qui le suit l'autre sur l'élément qui le précède, le parcours s'effectuant en suivant l'un ou l'autre sens de chaînage :



La classe **LinkedList** présente dans le package **java.util.LinkedList**, est en Java une implémentation de la liste chaînée bi-directionnelle, comme la classe Vector, les éléments de la classe **LinkedList** ne peuvent être que des objets et non de type élémentaires (byte, short, int, long ou char ne sont pas autorisés),

Quelques méthodes permettant de manipuler les éléments d'une LinkedList :

void addFirst(**Object** obj)	ajouter un élément au début de la liste

void addLast(**Object** obj)	ajouter un élément à la fin de la liste
void clear( )	effacer tous les éléments de la liste
Object get(**int** index)	élément situé au rang = 'index'
int indexOf(**Object** elem)	rang de l'élément 'elem'
Object remove(**int** index)	efface l'élément situé au rang = 'index'
Object set(**int** index , **Object** obj)	remplace l'élément de rang 'index' par obj
int size( )	nombre d'éléments de la liste

Reprenons l'exemple précédent sur une liste de type **LinkedList** d'éléments de type String :

```
import java.util.LinkedList;
class ApplicationLinkedList {

    static void afficheLinkedList (LinkedList  liste ) {
    //affiche une liste de chaînes
      System.out.println("liste taille = "+liste.size());
      for ( int i = 0 ; i <= liste.size( )-1 ; i++ )
           System.out.println("liste(" + i + ")=" + (String)liste.get(i));
    }
    static void LinkedListInitialiser( ) {
       LinkedList liste = new LinkedList( );
       String str = "val:";
       for ( int i = 0 ; i <= 5 ; i++ )
       liste.addLast( str + String.valueOf( i ) );
       afficheLinkedList(liste);
    }
    public static void main(String[] args) {
      LinkedListInitialiser( );
    }
   }
```

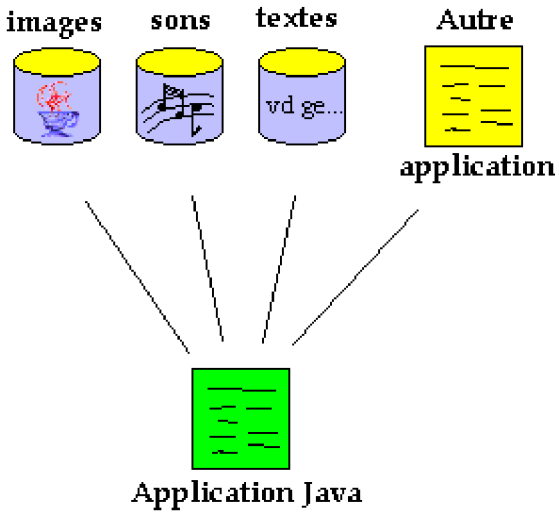
Voici le résultat de l'exécution de la méthode main de la classe ApplicationLinkedList **:**

```
liste taille = 6
liste(0) = val:0
liste(1) = val:1
liste(2) = val:2
liste(3) = val:3
liste(4) = val:4
liste(5) = val:5
```

Flux et fichiers

Java2

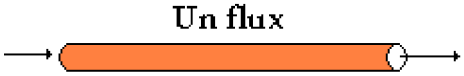
Un programme travaille avec ses données internes, mais habituellement il lui est très souvent nécessaire d'aller chercher en **entrée**, on dit **lire**, des nouvelles données (texte, image, son,...) en provenance de diverses sources (périphériques, autres applications...). Réciproquement, un programme peut après traitement, délivrer en **sortie** des résultats, on dit **écrire**, dans un fichier ou vers une autre application.



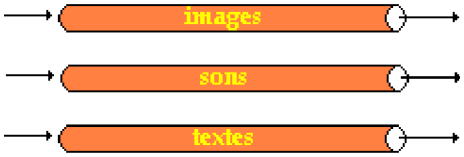
Les flux en Java

En Java, toutes ces données sont échangées en entrée et en sortie à travers des flux (Stream).

Un flux est une sorte de tuyau de transport séquentiel de données.



Il existe un flux par type de données à transporter :



Un flux est ***unidirectionnel*** : il y a donc des flux d'**entrée** et des flux de **sortie** :

Afin de jouer un son stocké dans un fichier, l'application Java ouvre en entrée, un flux associé aux sons et lit ce flux séquentiellement afin ensuite de traiter ce son (modifier ou jouer le son).

La même application peut aussi traiter des images à partir d'un fichier d'images et renvoyer ces images dans le fichier après traitement. Java ouvre un flux en entrée sur le fichier image et un flux en sortie sur le même fichier, l'application lit séquentiellement le flux d'entrée (octet par octet par exemple) et écrit séquentiellement dans le flux de sortie.

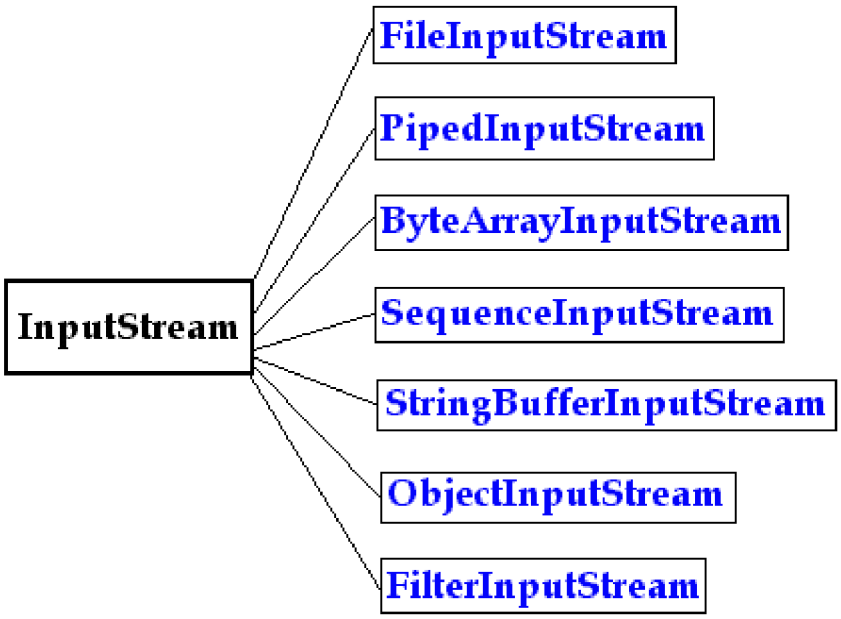
Java met à notre disposition dans le **package java.io.***, deux grandes catégories de flux :
(*la notation "*" dans* ***package java.io.**** *indique que l'on utilise toutes les classes du package java.io*)

- La famille des flux de caractères (caractères 16 bits)
- La famille des flux d'octets (information binaires sur 8 bits)

- Comme Java est un LOO (Langage Orienté Objet) les différents flux d'une famille sont des classes dont les méthodes sont adaptées au **transfert** et à la **structuration** des données selon la destination de celles-ci.
- Lorsque vous voulez lire ou écrire des données **binaires** (sons, images,...) utilisez une des classes de la famille des **flux d'octets**.
- Lorsque vous utilisez des données de type **caractères** préférez systématiquement l'un des classes de la famille des **flux de caractères**.

Etant donné l'utilité de tels flux nous donnons exhaustivement la liste et la fonction de chaque classe pour chacune des deux familles.

Les flux d'octets en entrée



Cette sous-famille de flux d'entrée contient 7 classes dérivant toutes de la classe abstraite InputStream.

Fonction des classes de flux d'octets en entrée

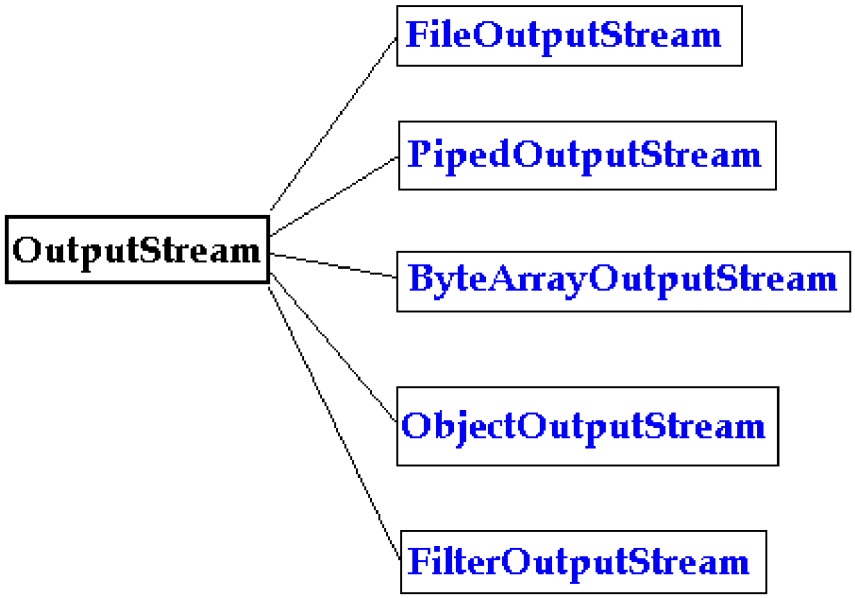
Classes utilisées pour la communication

FileInputStream	lecture de fichiers octets par octets.
PipedInputStream	récupère des données provenant d'un flux de sortie (cf. PipedOutputStream).
ByteArrayInputStream	lit des données dans un tampon structuré sous forme d'un array.

Classes utilisées pour le traitement

SequenceInputStream	concaténation d'une énumération de plusieurs flux d'entrée en un seul flux d'entrée.
StringBufferInputStream	lecture d'une String (Sun déconseille son utilisation et préconise son remplacement par **StringReader**).
ObjectInputStream	lecture d'objets Java.
FilterInputStream	lit à partir d'un InputStream quelconque des données, en "filtrant" certaines données.

Les flux d'octets en sortie



Cette sous-famille de flux de sortie contient 5 classes dérivant toutes de la classe abstraite **OutputStream**.

Fonction des classes de flux d'octets en sortie

Classes utilisées pour la communication

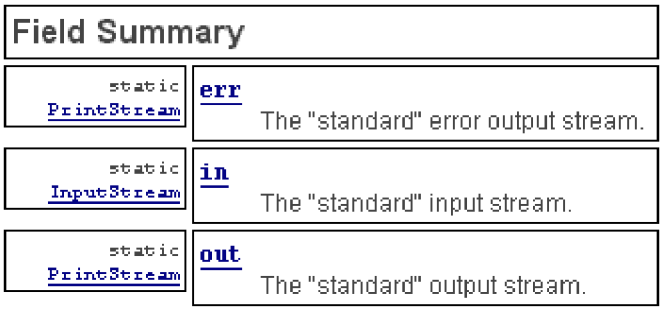
FileOutputStream	écriture de fichiers octets par octets.
PipedOutputStream	envoie des données vers un flux d'entrée (cf. PipedInputStream).
ByteArrayOutputStream	écrit des données dans un tampon structuré sous forme d'un array.

Classes utilisées pour le traitement

ObjectOutputStream	écriture d'objets Java lisibles avec ObjectInputStream.
FilterOutputStream	écrit à partir d'un OutputStream quelconque des données, en "filtrant" certaines données.

Les opérations d'entrée sortie standard dans une application

Java met à votre disposition 3 flux spécifiques présents comme attributs dans la classe System du package java.lang.System :



Le flux d'entrée **System.in** est connecté à l'entrée standard qui est par défaut le clavier.
Le flux de sortie **System.out** est connecté à la sortie standard qui est par défaut l'écran.
Le flux de sortie **System.err** est connecté à la sortie standard qui est par défaut l'écran.

La classe **PrintStream** dérive de la classe **FilterOutputStream**. Elle ajoute de nouvelles fonctionnalités à un flux de sortie, en particulier le flux **out** possède ainsi une méthode **println** redéfinies avec plusieurs signatures (plusieurs en-têtes différentes : byte, short, char, float,...) qui lui permet d'écrire des entiers de toute taille, des caractères, des réels...

Vous avez pu remarquer que depuis le début nous utilisions pour afficher nos résultats, l'instruction **System.out.println**(...); qui en fait correspond à l'utilisation de la méthode println de la classe **PrintStream**.

*Exemple d'utilisation simple des flux **System.out** et **System.in*** :

```
public static void main(String[] args) throws IOException {
      System.out.println("Appuyez sur la touche <Entrée> :"); //message écran
      System.in.read( ); // attend la frappe clavier de la touche <Entrée>
}
```

Dans Java, le flux **System.in** appartient à la classe **InputStream** et donc il est moins bien traité que le flux **System.out** et donc il n'y a pas en Java quelque chose qui ressemble à l'instruction **readln** du pascal par exemple. Le manque de souplesse semble provenir du fait qu'une méthode ne peut renvoyer son résultat de type élémentaire que par l'instruction **return** et il n'est pas possible de redéfinir une méthode uniquement par le type de son résultat.

Afin de pallier à cet inconvénient il vous est fourni (ou vous devez écrire vous-même) une classe **Readln** avec une méthode de lecture au clavier pour chaque type élémentaire. En mettant le fichier **Readln.class** dans le même dossier que votre application vous pouvez vous servir de cette classe pour lire au clavier dans vos programmes n'importe quelles variables de type élémentaire.

Méthodes de lecture clavier dans la classe Readln

String unstring( )	lecture clavier d'un chaîne de type String.
byte unbyte( )	lecture clavier d'un entier de type byte.
short unshort( )	lecture clavier d'un entier de type short.
int unint( )	lecture clavier d'un entier de type int.
long unlong( )	lecture clavier d'un entier de type long.

double undouble( )	lecture clavier d'un réel de type double.
float unfloat( )	lecture clavier d'un réel de type float.
char unchar( )	lecture clavier d'un caractère.

Voici un exemple d'utilisation de ces méthodes dans un programme :

```
class ApplicationLireClavier {

 public static void main(String [ ] argument)  {
 String Str;
 int i;  long L;  char k;
 short s;  byte b;  float f; double d;
  System.out.print("Entrez une chaîne : ");
  Str = Readln.unstring( );
  System.out.print("Entrez un int: ");
  i = Readln.unint( );
  System.out.print("Entrez un long : ");
  L = Readln.unlong( );
  System.out.print("Entrez un short : ");
  s = Readln.unshort( );
  System.out.print("Entrez un byte : ");
  b = Readln.unbyte( );
  System.out.print("Entrez un caractère : ");
  k = Readln.unchar( );
  System.out.print("Entrez un float : ");
  f = Readln.unfloat( );
  System.out.print("Entrez un double : ");
  f = Readln.unfloat( );
 }
}
```

Les flux de caractères

Cette sous-famille de flux de données sur 16 bits contient des classes dérivant toutes de la classe abstraite **Reader** pour les flux en entrée, et des classes relativement aux flux en sortie dérivant de la classe abstraite **Writer**.

Fonction des classes de flux de caractères en entrée

BufferedReader	lecture de caractères dans un tampon.
CharArrayReader	lit de caractères dans un tampon structuré sous forme d'un array.
FileReader	lecture de caractères dans un fichier texte.
FilterReader	lit à partir d'un Reader quelconque des caractères, en "filtrant" certaines caractères.

InputStreamReader	conversion de flux d'octets en flux de caractères (8 bits en 16 bits)
LineNumberReader	lecture de caractères dans un tampon (dérive de BufferedReader) avec comptage de lignes.
PipedReader	récupère des données provenant d'un flux de caractères en sortie (cf. PipedWriter).
StringReader	lecture de caractères à partir d'une chaîne de type String.

Fonction des classes de flux de caractères en sortie

BufferedWriter	écriture de caractères dans un tampon.
CharArrayWriterWriter	écrit des caractères dans un tampon structuré sous forme d'un array.
FileWriter	écriture de caractères dans un fichier texte.
FilterWriter	écrit à partir d'un Reader quelconque des caractères, en "filtrant" certaines caractères.
OutputStreamWriter	conversion de flux d'octets en flux de caractères (8 bits en 16 bits)
PipedWriter	envoie des données vers un flux d'entrée (cf. PipedReader).
StringWriter	écriture de caractères dans une chaîne de type String.

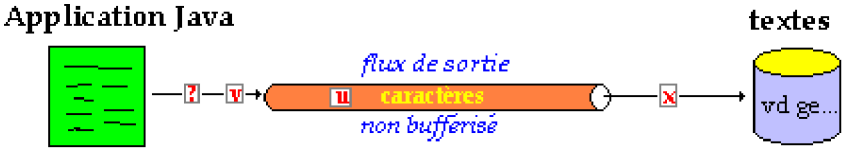
Lecture et écriture dans un fichier de texte

Il existe une classe dénommée **File** (dans **java.io.File**) qui est une représentation abstraite des fichiers, des répertoires et des chemins d'accès. Cette classe permet de créer un fichier, de l'effacer, de le renommer, de créer des répertoires etc...

Pour construire un fichier (texte ou non) il est nécessaire de le créer, puis d'y écrire des données à l'intérieur. Ces opérations passent obligatoirement en Java, par la connexion du fichier après sa création, à un flux de sortie. Pour utiliser un fichier déjà créé (présent sur disque local ou télétransmis) il est nécessaire de se servir d'un flux d'entrée.

Conseil pratique : pour tous vos fichiers utilisez systématiquement les flux d'entrée et de sortie bufférisés (**BufferedWriter** et **BufferedReader** par exemple, pour vos fichiers de textes). Dans le cas d'un flux non bufférisé le programme lit ou écrit par exemple sur le disque dur, les données au fur et à mesure, alors que les accès disque sont excessivement coûteux en temps.

Exemple écriture non bufférisée de caractères dans un fichier texte :



Ci-dessous un exemple de méthode permettant de créer un fichier de caractères et d'écrire une suite de caractères terminée par le caractère '#', on utilise un flux de la classe **FileWriter** non bufférisée :

Rappel :

FileReader	lecture de caractères dans un fichier texte.
FileWriter	écriture de caractères dans un fichier texte.

```
public static void fichierFileWriter(String nomFichier) {
  try {
    FileWriter out = new FileWriter(nomFichier);
    out.write("Ceci est une ligne FileWriter");
    out.write('#');
    out.close( );
  }
  catch (IOException err) {
    System.out.println( "Erreur : " + err );
  }
}
```

L'exécution de cette méthode produit le texte suivant :
Ceci est une ligne FileWriter#

Un flux **bufférisé** stocke les données dans un tampon (buffer, ou mémoire intermédiaire) en mémoire centrale, puis lorsque le tampon est plein, le flux transfert le paquet de données contenu dans le tampon vers le fichier (en sortie) ou en provenance du fichier en entrée. Dans le cas d'un disque dur, les temps d'accès au disque sont optimisés puisque celui-ci est moins fréquemment sollicité par l'écriture.

Exemple écriture bufférisée de caractères dans un fichier texte :

Ci-dessous un exemple de méthode permettant de créer un fichier de caractères et d'écrire une suite de caractères terminée par le caractère # , on utilise un flux de la classe **BufferedWriter**

bufférisée qui comporte la même méthode write, mais qui possède en plus la méthode newLine ajoutant un end of line (fin de ligne) à une suite de caractères, permettant le stockage simple de texte constitué de lignes:

Rappel :

BufferedReader	lecture de caractères dans un tampon.
BufferedWriter	écriture de caractères dans un tampon.

```
public static void fichierBufferedWriter(String nomFichier) {
  try {
    fluxwrite = new FileWriter(nomFichier);
    BufferedWriter out = new BufferedWriter(fluxwrite);
    out.write("Ceci est une ligne FileBuffWriter");
    out.write('#');
    out.write("Ceci est la ligne FileBuffWriter n° 1");
    out.newLine( ); //écrit le eoln
    out.write("Ceci est la ligne FileBuffWriter n° 2");
    out.newLine( ); //écrit le eoln
    out.close( );
  }
 catch (IOException err) {
   System.out.println( "Erreur : " + err );
 }
}
```

L'exécution de cette méthode produit le texte suivant :
Ceci est une ligne FileBuffWriter#Ceci est la ligne FileBuffWriter n° 1
Ceci est la ligne FileBuffWriter n° 2

Nous avons utilisé la déclaration de flux bufférisée explicite complète :

fluxwrite = **new** FileWriter(nomFichier); BufferedWriter out = **new** BufferedWriter(fluxwrite) ;	BufferedWriter out = **new** BufferedWriter (**new** FileWriter(nomFichier));