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: **( )**.

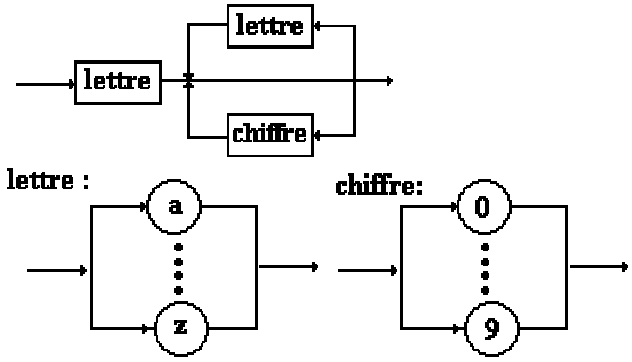
- **int** x ;
  x = **(int)** y ; signifie que vous demander de **transtyper** la valeur contenue dans la variable y en **un entier signé 32 bits** avant de la mettre dans la variable x.
- Tous les types élémentaires peuvent être transtypés à l'exception du type **boolean** qui ne peut pas être converti 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 :

□p □ { **V**, **F** } , □q □ { **V**, **F** } , p **&**q est toujours évalué en entier *(p et q sont toujours évalués)*.

□p □ { **V**, **F** } , □q □ { **V**, **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
□q □ { **V**, **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.

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

L'opérateur ou optimisé : | |

Théorème
□q □ { **V**, **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***

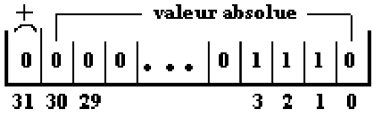
Afin de bien comprendre ces opérateurs, le lecteur doit bien connaître les différents codages des entiers en machine (binaire pur, binaire signé, complément à deux) car les entiers Java sont codés en complément à deux et la manipulation bit à bit nécessite une bonne compréhension de ce codage.

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

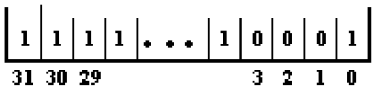
Les exemples en 4 instructions Java sur la même mémoire :

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

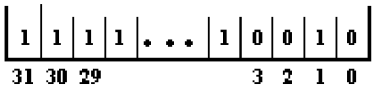
soit à représenter le nombre -14 dans la variable i de type **int** (entier signé sur 32 bits)



codage de |-14|= 14



complément à 1



addition de 1

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

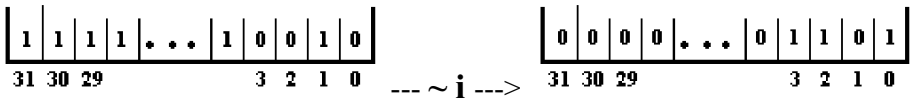
Soient la déclaration java suivante :

int i = -14 , j ;

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

- Etude de l'instruction : j = ~ i

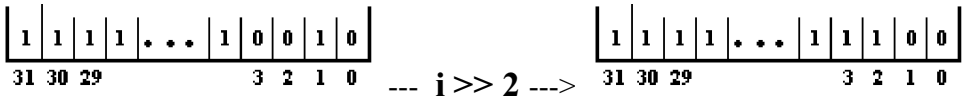
j = ~ i ; *// complémentation des bits de i*



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

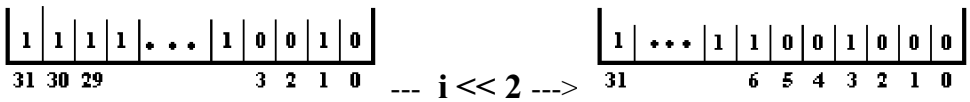
j = i >> 2 ; *// décalage avec signe de 2 bits vers la droite*



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*

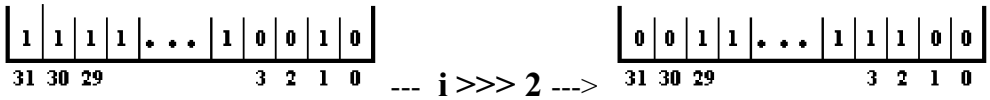


Tous les bits sont décalés de 2 positions vers la gauche (vers le bit de poids fort), des 0 sont introduits à partir de la droite (à partir du bit de poids faible) dans les emplacements libérés

(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 :

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

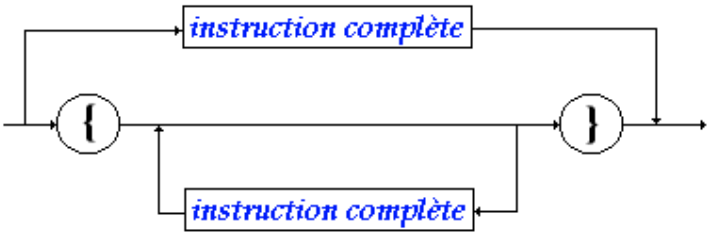
1 - les instructions de bloc

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 :

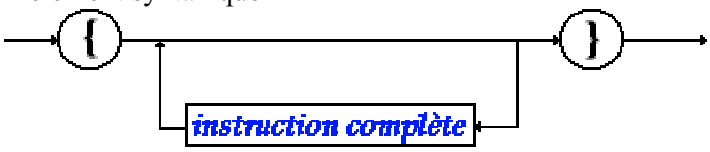


instruction complète :

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 ;
       }
    }
  }
```

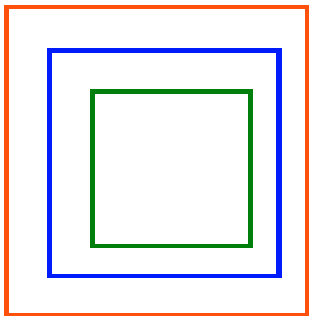


schéma d'imbrication des 3 blocs

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*

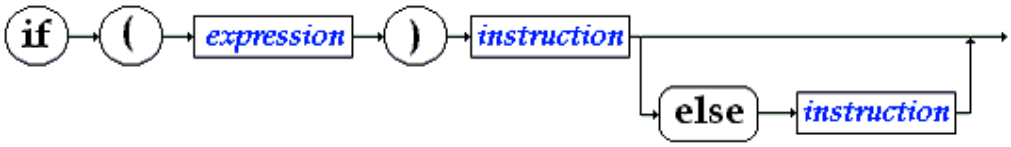
Les instructions conditionnelles

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) ;
```

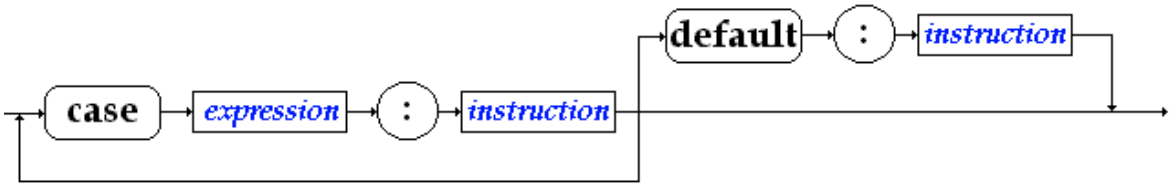
réponse *:*
a,b,c contiennent après exécution le plus grand des deux entiers contenus au départ dans a et b.

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).

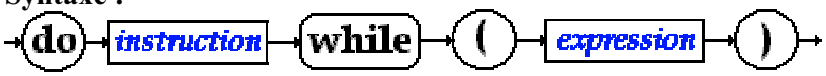
Sémantique :
Identique à celle du pascal (instruction algorithmique **tantque** .. **faire** .. **ftant**) avec le même défaut de fermeture de la boucle.

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** :

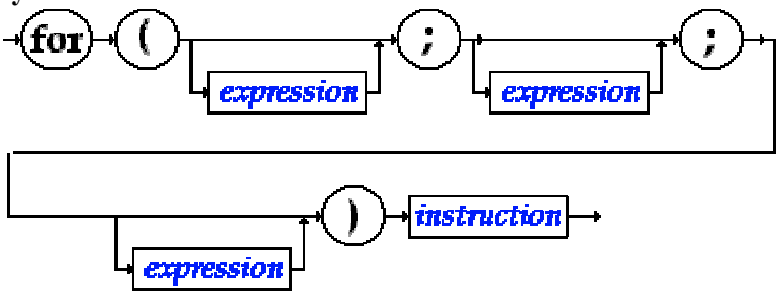
do Instr **while** (Expr) □□□ Instr ; **while** (Expr) Instr

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 Applicboucle_while {

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


}
```

```
class Applicboucle_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

….

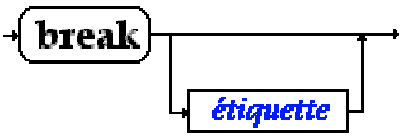
x = 9

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);
```

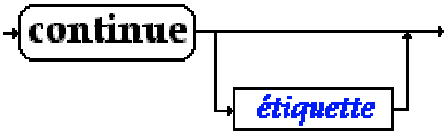
Explications

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.

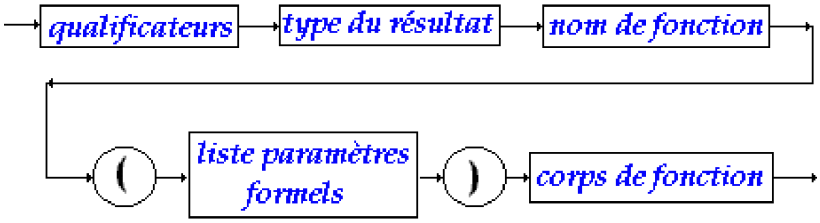
Une méthode de classe commence **obligatoirement** par le mot clef **static**.

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*.

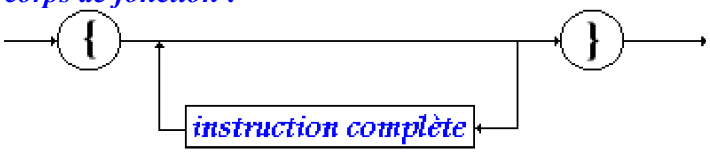
Déclaration d'une méthode

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 :

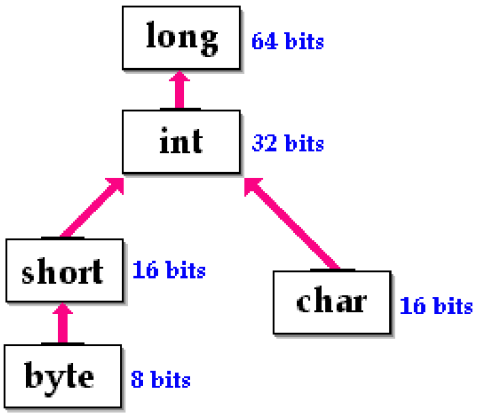
Dans tous les langages possédant la notion de sous-programme (ou fonction ou procédure), il se pose une question à savoir, les paramètres **formels** décrits lors de la déclaration d'un sous-programme ne sont que des variables muettes servant à expliquer le fonctionnement du sous-programme sur des futures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

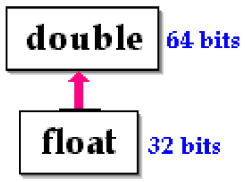
Compatibilité des types des paramètres

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.

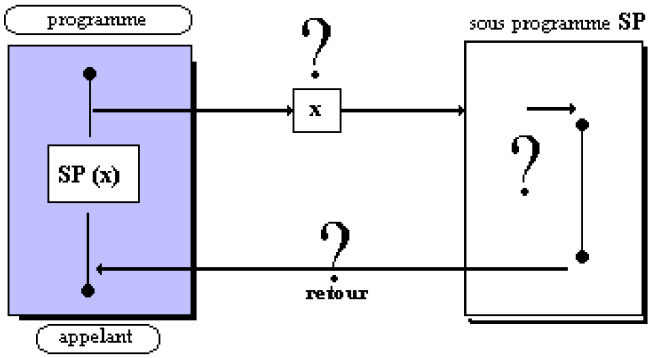
- Le paramètre effectif "**i**" est associé au paramètre formel **rang**.(**short** = entier signé sur 16 bits et **int** = entier signé sur 32 bits)

```
      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

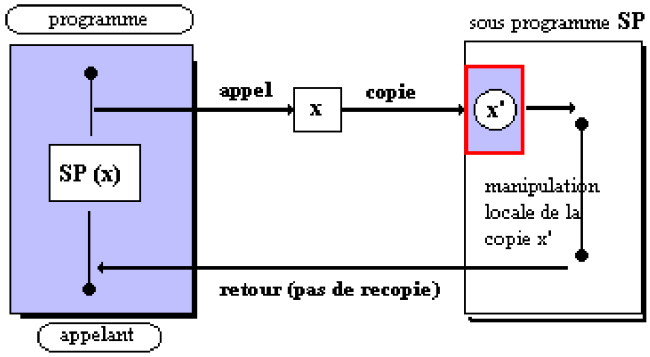
Un paramètre effectif transmis au sous-programme appelé est en fait un moyen d'utiliser ou d'accéder à une information appartenant au bloc appelant (le bloc appelé peut être le même que le bloc appelant, il s'agit alors de récursivité).



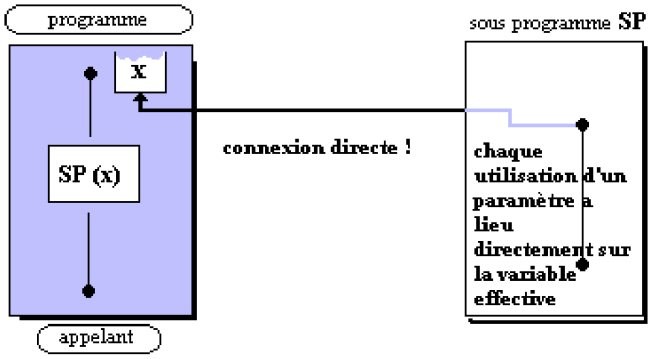
En Java, il existe deux modes de transmission (ou de passage) des paramètres (semblables à Delphi).

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 :

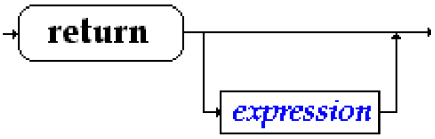
Le choix de **passage selon les types** élimine les inconvénients dûs à l'encombrement mémoire et à la lenteur de recopie de la valeur du paramètre par exemple dans un passage par valeur, car nous verrons plus loin que les **tableaux en Java sont des objets** et qu'ils sont donc passés **par référence**.

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

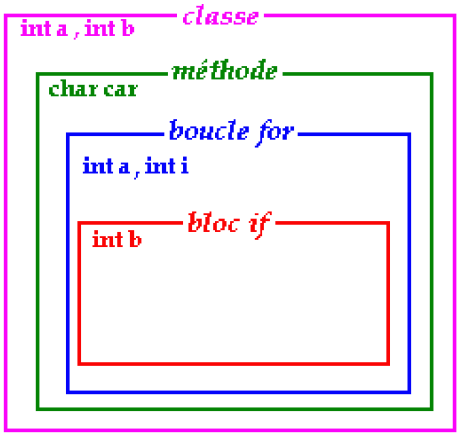
Les variables définies dans des blocs du genre instructions composées, boucles, try..catch ne sont visibles que dans le bloc et ses sous-blocs imbriqués, dans lequel elle sont définies.

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**.

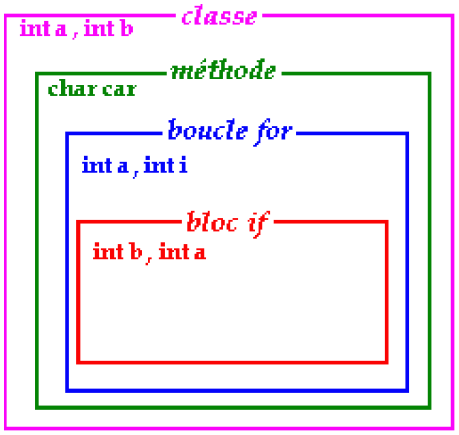
La variable de classe "b" définie dans **static int b** = 2; est masquée dans la méthode **f** dans le bloc imbriqué **if**.

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).

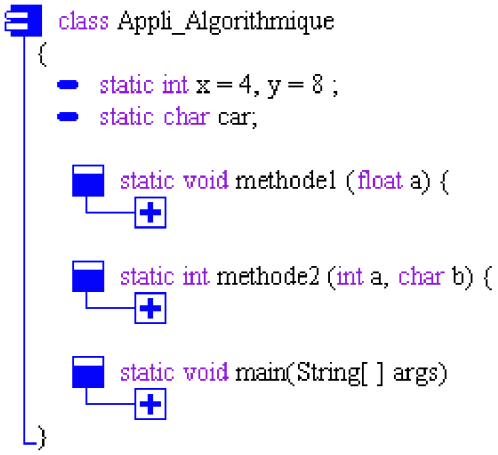
Synthèse : utiliser une méthode static dans une classe

❖ Les méthodes représentent les actions

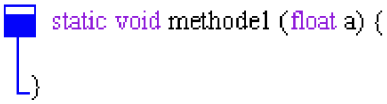
En POO, les méthodes d'une classe servent à indiquer comment fonctionne un objet dans son environnement d'exécution. Dans le cas où l'on se restreint à utiliser Java comme un langage algorithmique, la classe représentant le programme principal, les méthodes représenteront les sous programmes du programme 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)

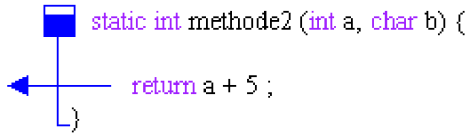


Les méthodes type procédure (méthode pouvant avoir plusieurs paramètres en entrée, mais ne renvoyant pas de résultat) sont précédées du mot clef **void** :



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

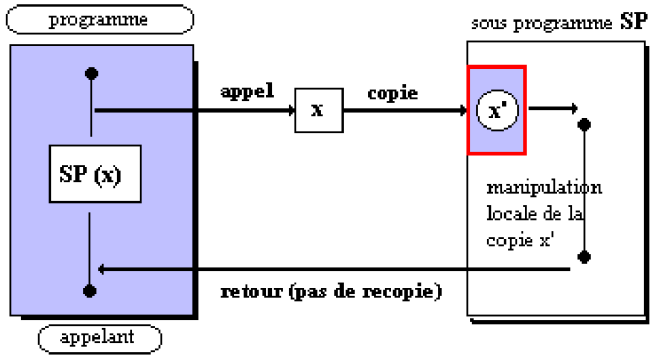
résultat du type déclaré par le mot clef précédant le nom de la fonction :



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 :



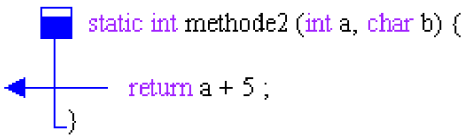
En Java tous les paramètres sont passés par valeur (même si l'on dit que **Java passe les objets par référence**, en fait il s'agit très précisément de **la référence de l'objet qui est passée par valeur**). Pour ce qui est de la vision algorithmique de Java, le passage par valeur est la norme.

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**

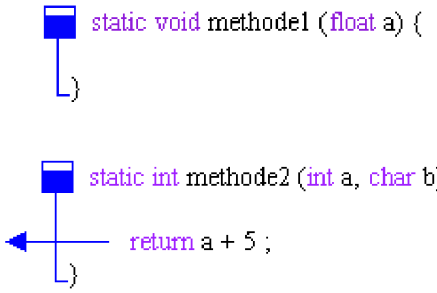


Les instructions ci-après permettent de modifier la valeur d'une variable x à l'aide de la méthode "methode2" :

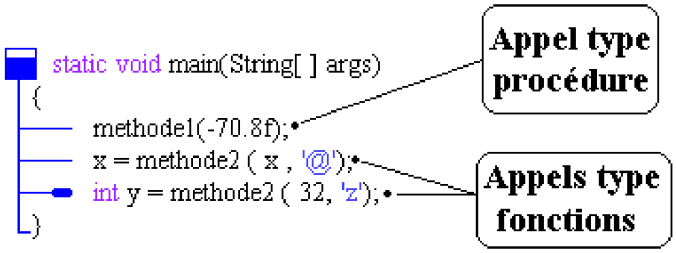
```
int x = 10;
   // avant l'appel x vaut 10
x = methode2 ( x , '@');
  // après l'appel x vaut 15
```

❖ Appeler une méthode en Java

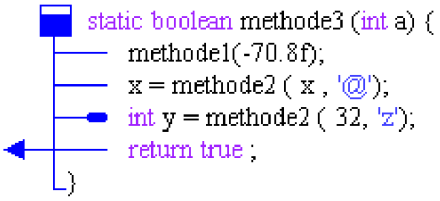
1. Soit on appelle la méthode comme une procédure en Pascal (avec ses paramètres effectifs éventuels), soit on l'appelle comme une fonction en utilisant son résultat. Reprenons les deux méthodes précédentes et appelons les :



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

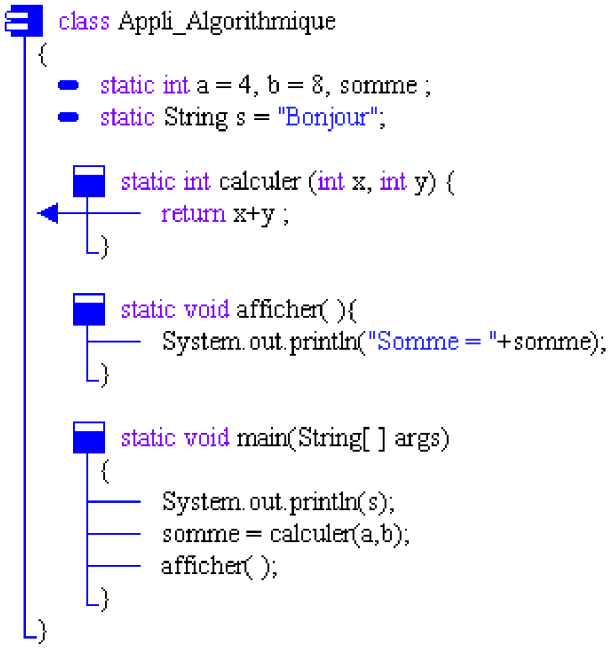


3. Soit on peut les appeler dans une autre méthode "methode3" :



❖ 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

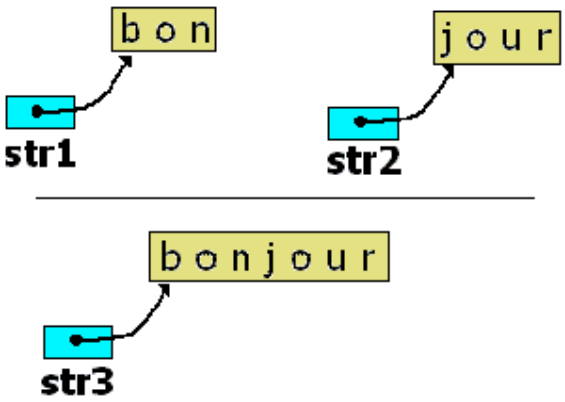
Un opérateur ou une méthode

Opérateur : + sur les chaînes

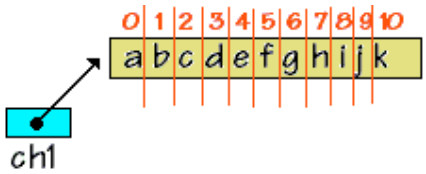
ou

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 :

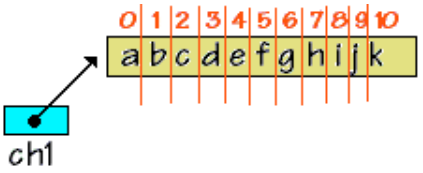
char charAt(int rang)

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 :

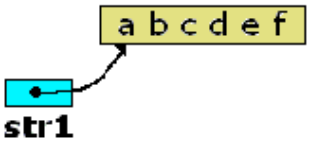
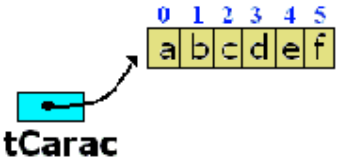
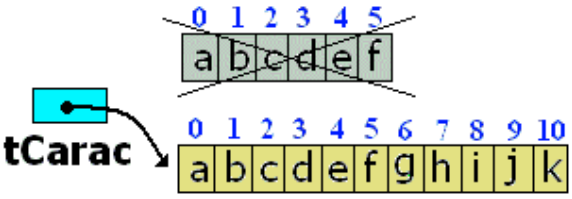
String str1 = "abcdef" ;

char [ ] tCarac ;

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

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

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

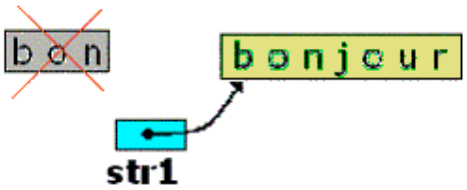
String str1 = "abcdef" ;	 str1 référence un objet de chaîne.
char [ ] tCarac ; tCarac = str1.**toCharArray**( ) ;	 tCarac référence un objet de tableau à 6 éléments.
tCarac = "abcdefghijk".**toCharArray**( );	 tCarac référence maintenant un nouvel objet de tableau à 11 éléments, l'objet précédent est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

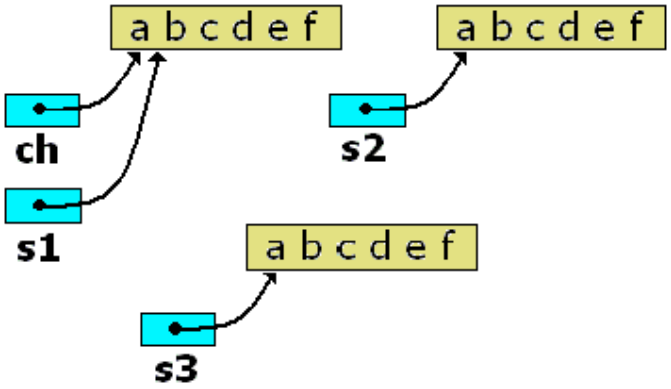
- La méthode **boolean** equals(Object s) teste si deux chaînes n'ayant pas la même référence ont la même valeur.

  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

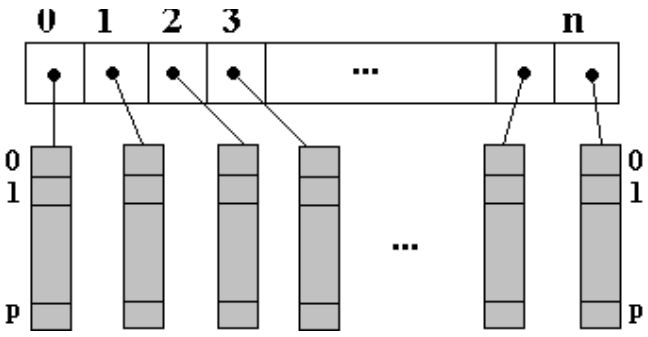
Les tableaux en Java peuvent être à deux dimensions (voir même à plus de deux) auquel cas on peut les appeler des matrices, ce sont aussi des objets et ils se comportent comme les tableaux à une dimension tant au niveau des déclarations qu'au niveau des utilisations. La déclaration s'effectue avec deux 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.

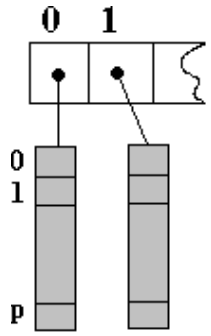
Un tel tableau à deux dimensions, peut être considéré comme un tableau unidimensionnel de pointeurs, où chaque pointeur référence un autre tableau unidimensionnel.

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

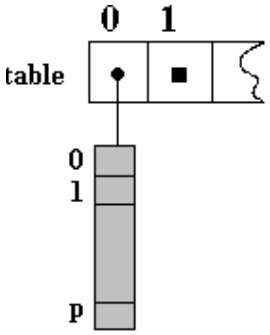
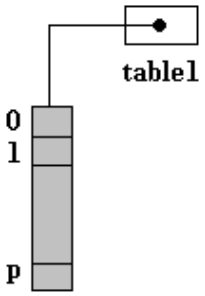
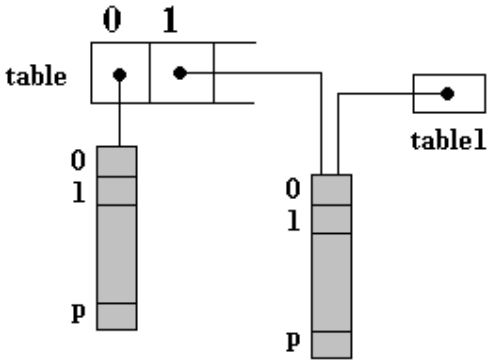
int [ ][ ] table = **new int** [n+1][p+1];



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 :

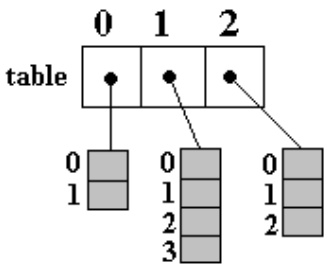
table[0] = **new** int[p+1]; table[1] = **new** int[p+1]; Dans chacune de ces deux instructions nous créons un objet de tableau unidimensionnel qui est référencé par la cellule de rang 0, puis par celle de rang 1.	

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.

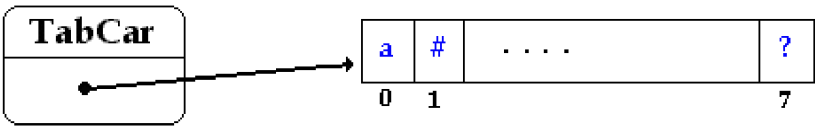
Tableaux dynamiques et listes

Java2

Tableau dynamique

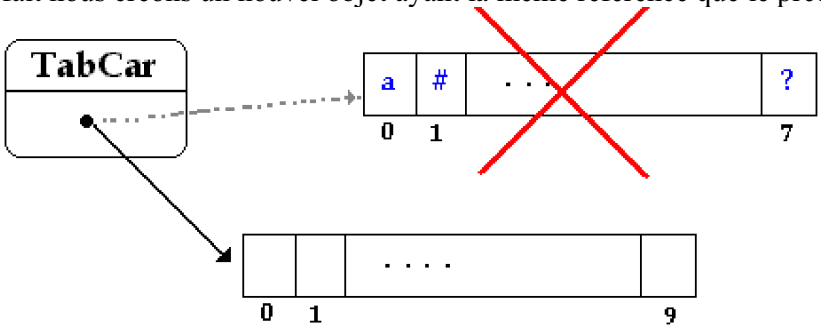
Un tableau array à une dimension, lorsque sa taille a été fixée soit par une définition explicite, soit par une définition implicite, **ne peut plus changer de taille**, c'est donc une structure statique.

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);
}
```

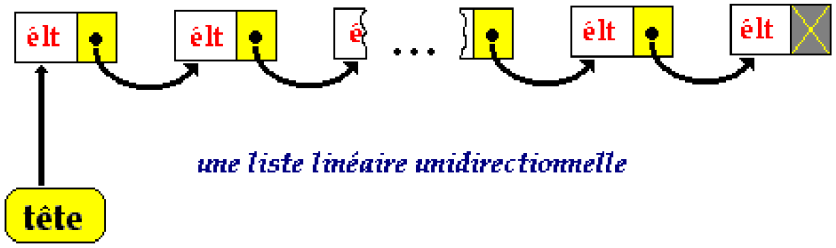
*Voici le résultat de l'exécution de la méthode***VectorInitialiser *:***

```
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
```

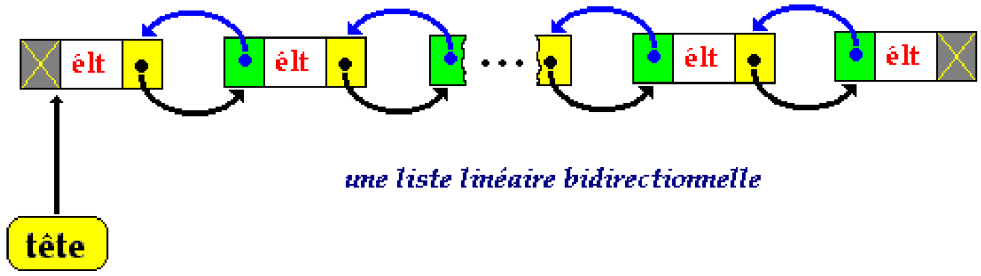
Les listes chaînées

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( );
    }
   }
```

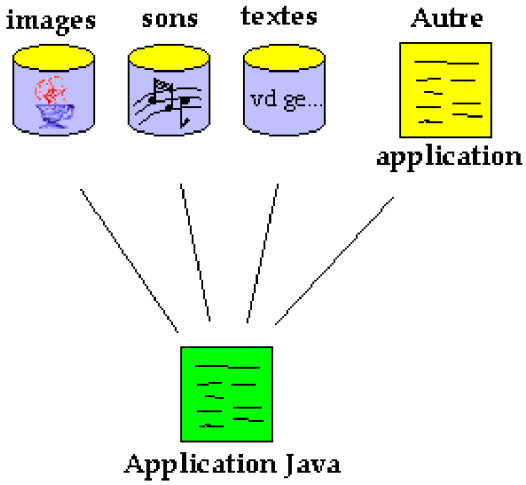
Voici le résultat de l'exécution de la méthode main de la classe ApplicationLinkedList ***:***

```
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

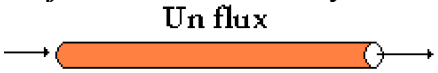
Un programme travaille avec ses données internes, mais habituellement il lui est très souvent nécessaire d'aller chercher en **entrée**, on dit **lire**, des nouvelles données (texte, image, son,...) en provenance de diverses sources (périphériques, autres applications...). Réciproquement, un programme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



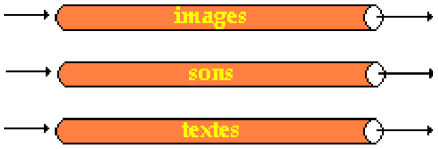
Les flux en Java

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.

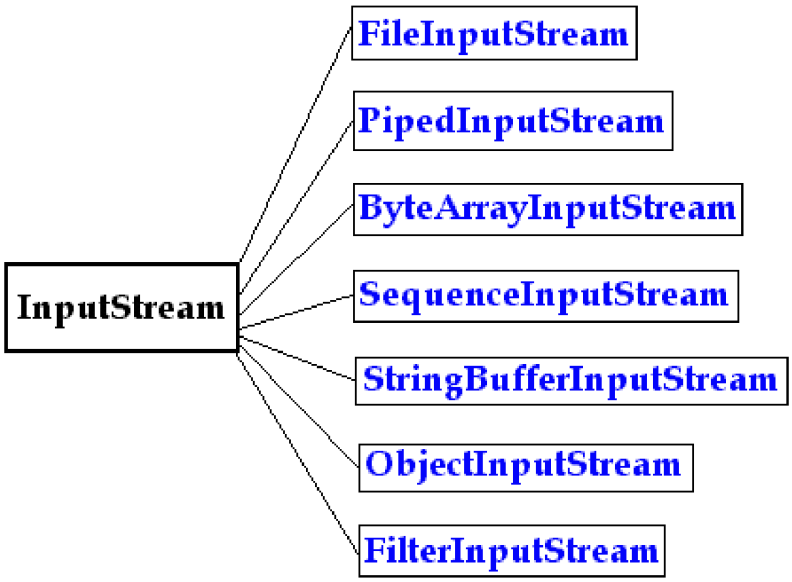
Java met à notre disposition dans le **package java.io.***, deux grandes catégories de flux :
(*la notation "*" dans* ***package java.io.**** *indique que l'on utilise toutes les classes du package java.io*)

- 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

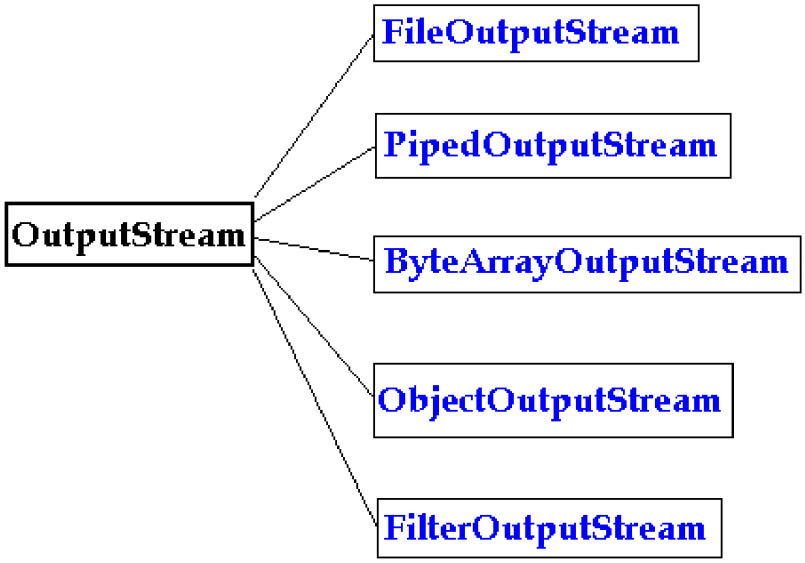
Classes utilisées pour la communication

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

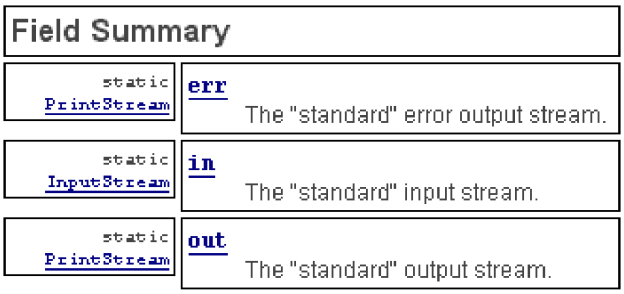
FileOutputStream	écriture de fichiers octets par octets.
PipedOutputStream	envoie des données vers un flux d'entrée (cf. PipedInputStream).
ByteArrayOutputStream	écrit des données dans un tampon structuré sous forme d'un array.

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.

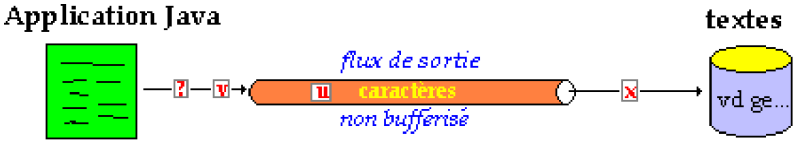
Lecture et écriture dans un fichier de texte

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));

Java langage orienté objet

Le contenu de ce thème :

Les classes

Les objets

Les membres : attributs et méthodes

les interfaces

Java2 à la fenêtre - avec Awt

exercicesJava2 IHM - Awt

IHM - avec Swing

exercices IHM - JFrame de Swing

Les applets Java

Afficher des composants, redessiner une Applet

Les classes

Java2

Nous proposons des comparaisons entre les syntaxes Delphi et Java lorsque les définitions sont semblables.

Les classes : des nouveaux types

Rappelons un point fondamental déjà indiqué : tout programme Java du type **application** ou **applet** contient une ou plusieurs classes précédées ou non d'une déclaration d'importation de classes contenues dans des bibliothèques (clause **import**) ou à un package complet composé de nombreuses classes. La notion de module en Java est représentée par le **package**.

Delphi	Java
Unit Biblio; **interface** *// les déclarations des classes* **implementation** *// les implémentations des classes* **end**.	**package** Biblio; *// les déclarations et implémentation des classes*

Déclaration d'une classe

En Java nous n'avons pas comme en Delphi, une partie déclaration de la classe et une partie implémentation séparées l'une de l'autre. La classe avec ses attributs et ses méthodes sont déclarés et implémentés à un seul endroit.

Delphi	Java
interface **uses** biblio; **type** Exemple = **class** x : real; y : integer; **function** F1(a,b:integer): real;	**import** biblio; **class** Exemple { **float** x; **int** y; **float** F1(**int** a, **int** b)

```
  procedure P2;
end;

implementation
 function F1(a,b:integer): real;
 begin
  .....code de F1
 end;
 procedure P2;
 begin
  .....code de P2
 end;
end.
```

```
  {
    .....code de F1
  }

  void P2( )
  {
    .....code de P2
  }
 }
```

Une classe est un type Java

Comme en Delphi, une classe Java peut être considérée comme un nouveau type dans le programme et donc des variables d'objets peuvent être déclarées selon ce nouveau "type".

Une déclaration de programme comprenant 3 classes en Delphi et Java :

Delphi

```
interface
type
   Un = class
        ...
   end;

    Deux = class
      ...
    end;

    Appli3Classes = class
      x : Un;
      y : Deux;
     public
       procedure main;
    end;

    implementation

    procedure Appli3Classes.main;
    var
      x : Un;
      y : Deux;
    begin
      ...
    end;

   end.
```

Java

```
class Appli3Classes
{   Un x;
     Deux y;
  public static void main(String [ ] arg) {
    Un x;
    Deux y;
    ...
  }
 }
 class Un
 { ...
 }

  class Deux
  { ...
  }
```

Toutes les classes ont le même ancêtre - héritage

Comme en Delphi toutes les classes Java dérivent automatiquement d'une seule et même classe ancêtre : la classe **Object**. En java le mot-clef pour indiquer la dérivation (héritage) à partir d'une autre classe est le mot **extends**, lorsqu'il est omis c'est donc que la classe hérite automatiquement de la classe **Object** :

Les deux déclarations de classe ci-dessous sont équivalentes en Delphi et en Java

Delphi	Java
type Exemple = **class** (TObject) ..... **end**;	**class** Exemple **extends** Object { ..... }
type Exemple = **class** ..... **end**;	**class** Exemple { ..... }

L'héritage en Java est classiquement de l'**héritage simple** comme en Delphi. Une classe fille qui dérive (on dit qui étend en Java) d'une seule classe mère, hérite de sa classe mère toutes ses méthodes et tous ses champs. En Java la syntaxe de l'héritage fait intervenir le mot clef **extends**, comme dans "**class** Exemple **extends** Object".

Une déclaration du type :

```
class ClasseFille extends ClasseMere {
}
```

signifie que la classe ClasseFille dispose de tous les attributs et les méthodes de la classe ClasseMere.

Comparaison héritage Delphi et Java :

Delphi	Java
type ClasseMere = **class** // champs de ClasseMere // méthodes de ClasseMere **end**; ClasseFille = **class (** ClasseMere **)** // hérite des champs de ClasseMere // hérite des méthodes de ClasseMere **end**;	**class** ClasseMere { // champs de ClasseMere // méthodes de ClasseMere } **class** ClasseFille **extends** ClasseMere { // hérite des champs de ClasseMere // hérite des méthodes de ClasseMere }

Bien entendu une classe fille peut définir de nouveaux champs et de nouvelles méthodes qui lui sont propres.

Encapsulation des classes

La visibilité et la protection des classes en Delphi, est apportée par le module **Unit** où toutes les classes sont visibles dans le module en entier et dès que la unit est utilisée les classes sont visibles partout. Il n'y a pas de possibilité d'imbriquer une classe dans une autre.

En Java depuis le JDK 1.1, la situation qui était semblable à celle de Delphi a considérablement évolué et actuellement en Java 2, nous avons la possibilité d'***imbriquer*** des classes dans d'autres classes, par conséquent la ***visibilité de bloc s'applique aussi aux classes.***

Mots clefs pour la protection des classes et leur visibilité :

- Une classe Java peut se voir attribuer un modificateur de comportement sous la forme d'un mot clef devant la déclaration de classe.Par défaut si aucun mot clef n'est indiqué la classe est visible dans tout le package dans lequel elle est définie (si elle est dans un package). Il y a 2 mots clefs possibles pour modifier le comportement d'une classe : **public** et **abstract**.

Java	Explication
mot clef **abstract** : **abstract class** ApplicationClasse1 { ... }	classe abstraite **non instanciable**. Aucun objet ne peut être créé.
mot clef **public** : **public class** ApplicationClasse2 { ... }	classe visible **par n'importe quel** programme, elle doit avoir le même nom que le fichier de bytecode xxx.class qui la contient
pas de mot clef : **class** ApplicationClasse3 { ... }	classe visible **seulement** par toutes les autres classes du module où elle est définie.

Nous remarquons donc qu'une classe dès qu'elle est déclarée est toujours visible et qu'il y a en fait deux niveaux de visibilité selon que le modificateur **public** est, ou n'est pas présent, le mot clef **abstract** n'a de l'influence que pour l'héritage.

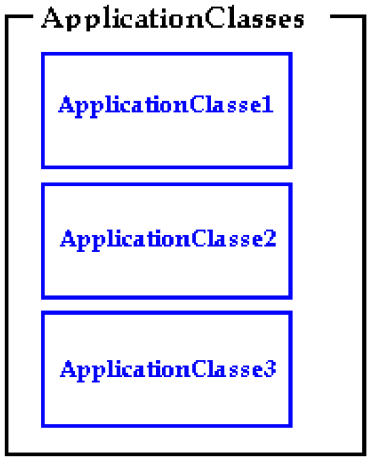
Nous étudions ci-après la visibilité des 3 classes précédentes dans deux contextes différents.

Exemple de classe intégrée dans une autre classe

Dans le premier contexte, ces trois classes sont utilisées en étant **intégrées** (imbriquées) à une

classe publique.

Exemple correspondant à l'imbrication de bloc suivante :



La classe :

Java	Explication
package Biblio; **public class** ApplicationClasses **{** **abstract class** ApplicationClasse1 { ... } **public class** ApplicationClasse2 { ... } **class** ApplicationClasse3 { ... } **}**	Ces 3 "sous-classes" sont visibles à partir de l'accès à la classe englobante "ApplicationClasses", elles peuvent donc être utilisées dans tout programme qui utilise la classe "ApplicationClasses".

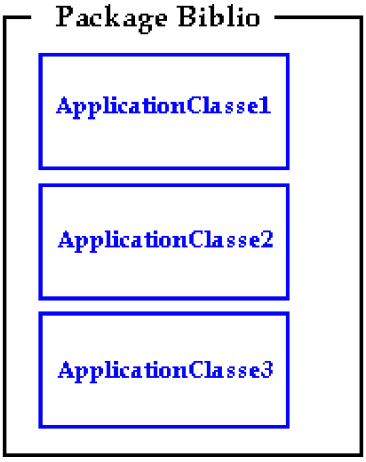
Un programme utilisant la classe :

Java	Explication
import Biblio.ApplicationClasses; **class** AppliTestClasses**{** ApplicationClasses.ApplicationClasse1 a1; ApplicationClasses.ApplicationClasse2 a2; ApplicationClasses.ApplicationClasse3 a3; **}**	Le programme de gauche "**class** AppliTestClasses" importe (*utilise*) la classe précédente ApplicationClasses et ses sous-classes, en déclarant 3 variables a1, a2, a3. La notation uniforme de chemin de classe est standard.

Exemple de classe inclue dans un package

Dans le second exemple, ces mêmes classes sont utilisées en étant **inclues** dans un package.

Exemple correspondant à l'imbrication de bloc suivante :



Le package :

Java	Explication
package Biblio; **abstract class** ApplicationClasse1 { ... } **public class** ApplicationClasse2 { ... } **class** ApplicationClasse3 { ... }	Ces 3 "sous-classes" font partie du package Biblio, elles sont visibles par importation séparée (comme précédemment) ou globale du package.

Un programme utilisant le package :

Java	Explication
import Biblio.* ; **class** AppliTestClasses{ ApplicationClasse1 a1; ApplicationClasse2 a2; ApplicationClasse3 a3; }	Le programme de gauche "**class** AppliTestClasses" importe le package Biblio et les classes qui le composent. Nous déclarons 3 variables a1, a2, a3.

Remarques pratiques :

Pour pouvoir utiliser dans un programme, une classe définie dans un module (**package**) celle-ci doit obligatoirement avoir été déclarée dans le **package**, avec le modificateur **public**.

Pour accéder à la classe Cl1 d'un **package** Pack1, il est nécessaire d'importer cette classe ainsi :

import Pack1.Cl1;

Méthodes abstraites

Le mot clef **abstract** est utilisé pour représenter **une classe ou une méthode abstraite**. Quel est l'intérêt de cette notion ? Le but est d'avoir des modèles génériques permettant de définir ultérieurement des actions spécifiques.

Une méthode déclarée en abstract dans une classe mère :

- N'a pas de corps de méthode.
- N'est pas exécutable.
- Doit obligatoirement être redéfinie dans une classe fille.

Une méthode **abstraite** n'est qu'une **signature** de méthode sans implémentation dans la classe.

Exemple de méthode abstraite :

```
class Etre_Vivant {
}
```

La classe Etre_Vivant est une classe mère générale pour les êtres vivants sur la planète, chaque catégorie d'être vivant peut être représenté par une classe dérivée (classe fille de cette classe) :

```
class Serpent extends Etre_Vivant {
}

class Oiseau extends Etre_Vivant {
}

class Homme extends Etre_Vivant {
}
```

Tous ces êtres se déplacent d'une manière générale donc une méthode SeDeplacer est commune à toutes les classes dérivées, toutefois chaque espèce exécute cette action d'une manière différente et donc on ne peut pas dire que se déplacer est une notion concrète mais une notion abstraite que chaque sous-classe précisera concrètement.

```
abstract class Etre_Vivant {
 abstract void SeDeplacer( );
 }

class Serpent extends Etre_Vivant {
```

```
  void SeDeplacer( ) {
    //.....en rampant
  }
}

class Oiseau extends Etre_Vivant {
  void SeDeplacer( ) {
    //.....en volant
  }
}

class Homme extends Etre_Vivant {
  void SeDeplacer( ) {
    //.....en marchant
  }
}
```

Comparaison de déclaration d'abstraction de méthode en Delphi et Java :

Delphi	Java
type Etre_Vivant = **class** **procedure** SeDeplacer;**virtual**;**abstract**; **end**; Serpent = **class** (Etre_Vivant) **procedure** SeDeplacer;**override**; **end**; Oiseau = **class** (Etre_Vivant) **procedure** SeDeplacer;**override**; **end**; Homme = **class** (Etre_Vivant) **procedure** SeDeplacer;**override**; **end**;	**abstract class** Etre_Vivant { **abstract void** SeDeplacer( ); } **class** Serpent **extends** Etre_Vivant { **void** SeDeplacer( ) { //.....en rampant } } **class** Oiseau **extends** Etre_Vivant { **void** SeDeplacer( ) { //.....en volant } } **class** Homme **extends** Etre_Vivant { **void** SeDeplacer( ) { //.....en marchant } }

En Delphi une méthode **abstraite** est une méthode **virtuelle** ou **dynamique** n'ayant pas d'implémentation dans la classe où elle est déclarée. Son implémentation est déléguée à une classe dérivée. Les méthodes abstraites doivent être déclarées en spécifiant la directive **abstract** après **virtual** ou **dynamic**.

Classe abstraite

Les classes abstraites permettent de créer des classes génériques **expliquant certains comportements sans les implémenter** et fournissant une implémentation commune de certains autres comportements pour l'héritage de classes. Les classes abstraites sont un outil intéressant pour le **polymorphisme**.

Vocabulaire et concepts :

- Une classe abstraite est une classe qui **ne** peut **pas** être instanciée.
- Une classe abstraite peut contenir des méthodes déjà implémentées.
- Une classe abstraite peut contenir des méthodes **non** implémentées.
- Une classe abstraite est héritable.
- On peut contsruire une hiérarchie de classes abstraites.
- Pour pouvoir construire un objet à partir d'une classe abstraite, il faut dériver une classe non abstraite en une classe implémentant **toutes** les méthodes **non** implémentées.

Une méthode déclarée dans une classe, **non implémentée** dans cette classe, mais juste définie par la déclaration de sa signature, est dénommée **méthode abstraite**.

Une méthode abstraite est une méthode à liaison dynamique n'ayant pas d'implémentation dans la classe où elle est déclarée. L' implémentation d'une méthode abstraite est déléguée à une classe dérivée.

Syntaxe de l'exemple en Delphi et en Java (C# est semblable à Delphi) :

Delphi	Java
Vehicule = **class** **public** **procedure** Demarrer; **virtual**;**abstract**; **procedure** RépartirPassagers; **virtual**; **procedure** PériodicitéMaintenance; **virtual**; **end;**	**abstract class** ClasseA { **public abstract void** Demarrer( ); **public void** RépartirPassagers( ); **public void** PériodicitéMaintenance( ); }

Si une classe contient au moins une méthode **abstract**, elle doit impérativement être déclarée en classe **abstract** elle-même.

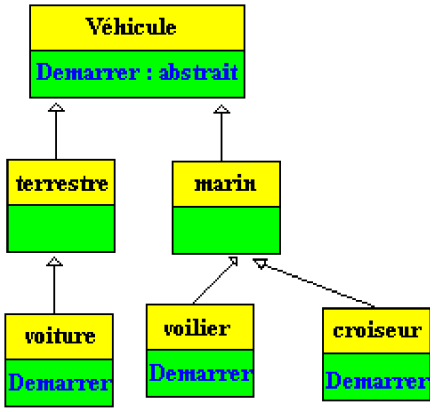
```
abstract class Etre_Vivant {
 abstract void SeDeplacer( );
}
```

Remarque

Une classe **abstract** ne peut pas être instanciée directement, seule une classe dérivée (sous-classe) qui redéfinit obligatoirement toutes les méthodes **abstract** de la classe mère peut être instanciée.

Conséquence de la remarque précédente, une classe dérivée qui redéfinit toutes les méthodes **abstract** de la classe mère sauf une (ou plus d'une) ne peut pas être instanciée et suit la même règle que la classe mère : elle contient au moins une méthode abstraite donc elle aussi une classe abstraite et doit donc être déclarée en **abstract**.

Si vous voulez utiliser la notion de classe abstraite pour fournir un polymorphisme à un groupe de classes, elles doivent toutes hériter de cette classe, comme dans l'exemple ci-dessous :

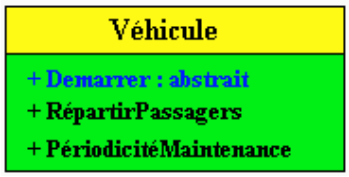


- La classe **Véhicule** est abstraite, car la méthode **Démarrer** est abstraite et sert de "modèle" aux futurs classes dérivant de **Véhicule**, c'est dans les classes **voiture**, **voilier** et **croiseur** que l'on implémente le comportement précis du genre de démarrage.

- Notons au passage que dans la hiérarchie précédente, les classes vehicule **Terrestre** et **Marin** héritent de la classe **Véhicule**, mais n'implémentent pas la méthode abstraite **Démarrer**, ce sont donc par construction des classes abstraites elles aussi.

Les classes abstraites peuvent également **contenir des membres déjà implémentés**. Dans cette éventualité, une classe abstraite propose un certain nombre de **fonctionnalités identiques** pour tous ses futurs descendants.(*ceci n'est pas possible avec une interface*).

Par exemple, la classe abstraite Véhicule n'implémente pas la méthode abstraite **Démarrer**, mais fournit et implante une méthode "**RépartirPassagers**" de répartition des passagers à bord du véhicule (fonction de la forme, du nombre de places, du personnel chargé de s'occuper de faire fonctionner le véhicule...), elle fournit aussi et implante une méthode "**PériodicitéMaintenance**" renvoyant la périodicité de la maintenance obligatoire du véhicule (fonction du nombre de kms ou miles parcourus, du nombre d'heures d'activités,...)



Ce qui signifie que toutes les classes **voiture**, **voilier** et **croiseur** savent comment répartir leurs éventuels passagers et quand effectuer une maintenance, chacune d'elle implémente son propre comportement de démarrage.

Dans cet exemple, supposons que :

Les classes **Vehicule**, **Marin** et **Terrestre** sont abstraites car aucune n'implémente la méthode abstraite **Demarrer**.

Les classes **Marin** et **Terrestre** contiennent chacune une surcharge dynamique implémentée de la méthode virtuelle PériodicitéMaintenance qui est déjà implémentée dans la classe Véhicule.

Les classes **Voiture**, **Voilier** et **Croiseur** ne sont pas abstraites car elles implémentent les (la) méthodes abstraites de leurs parents et elles surchargent dynamiquement (redéfinissent) la méthode virtuelle RépartirPassagers qui est implémentée dans la classe Véhicule.

Implantation d'un squelette Java de l'exemple

Java

```
abstract class Vehicule  {
  public abstract void Demarrer( );
  public void RépartirPassagers( ) { …  }
  public void PériodicitéMaintenance( ){ …  }
}
abstract class Terrestre extends Vehicule  {
  public void PériodicitéMaintenance( ) { … }
}
abstract class Marin extends Vehicule  {
  public void PériodicitéMaintenance( ) { … }
}
class Voiture extends Terrestre {
   public void Demarrer( ) { …  }
```

```
    public void RépartirPassagers( ) { … }
}
class Voilier extends Marin {
    public void Demarrer( ) { … }
    public void RépartirPassagers( ) { … }
}
class Croiseur extends Marin {
    public void Demarrer( ) { … }
    public void RépartirPassagers( ) { … }
}
```

Les objets

Java2

Les objets : des références

Les classes sont des descripteurs d'objets, les objets sont les agents effectifs et "vivants" implantant les actions d'un programme. Les objets dans un programme ont une vie propre :

- Ils naissent (ils sont créés ou alloués).
- Ils agissent (ils s'envoient des messages grâce à leurs méthodes).
- Ils meurent (ils sont désalloués, automatiquement en Java).

C'est dans le segment de mémoire de la machine virtuelle Java que s'effectue l'allocation et la désallocation d'objets. Le principe d'allocation et de représentation des objets en Java est identique à celui de Delphi il s'agit de la référence, qui est une encapsulation de la notion de pointeur.

Modèle de la référence et machine Java

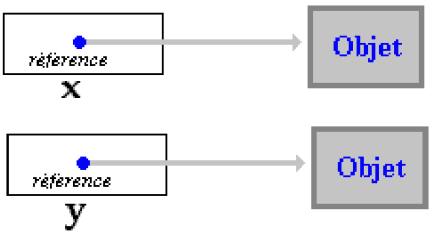
Rappelons que dans le modèle de la référence chaque objet (représenté par un identificateur de variable) est caractérisé par un couple (référence, bloc de données). Comme en Delphi, Java décompose l'**instanciation** (allocation) d'un objet en deux étapes :

- La déclaration d'identificateur de variable typée qui contiendra la référence,
- la création de la structure de données elle-même (bloc objet de données) avec **new**.

Delphi	Java
type **Un** = **class** ..... **end**; *// la déclaration :* **var**	**class Un** { ... } *// la déclaration :* **Un x , y ;**

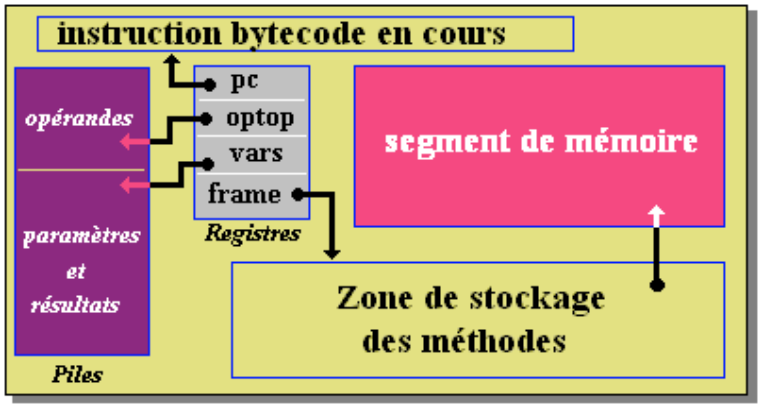
x , y : **Un**; .... *// la création :* **x** := Un.create ; **y** := Un.create ;	.... *// la création :* **x** = **new Un**( ); **y** = **new Un**( );

Après exécution du pseudo-programme précédent, les variables x et y contiennent chacune une référence (adresse mémoire) vers un bloc objet différent:

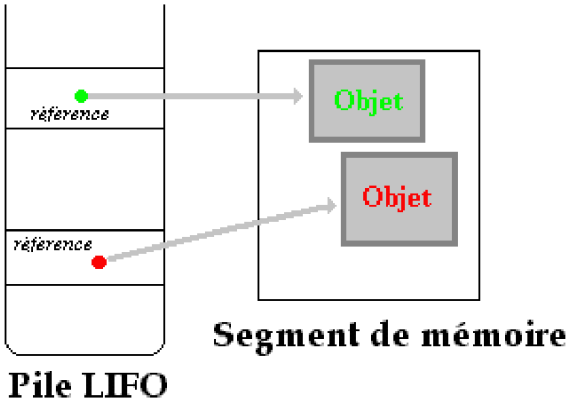


Un programme Java est fait pour être exécuté par une **machine virtuelle Java**, dont nous rappellons qu'elle contient 6 éléments principaux :

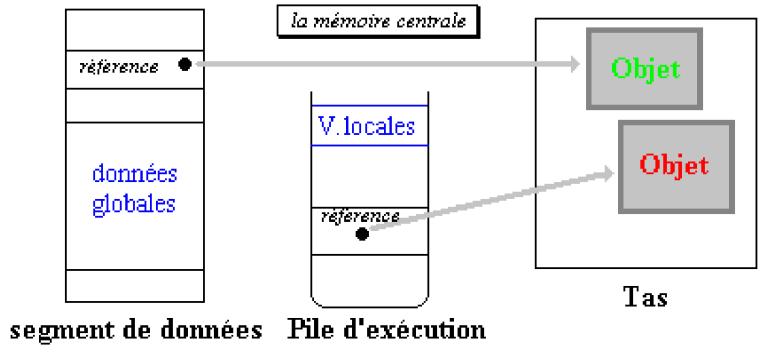
- Un jeu d'instructions en pseudo-code
- Une pile d'exécution LIFO utilisée pour stocker les paramètres des méthodes et les résultats des méthodes
- Une file FIFO d'opérandes pour stocker les paramètres et les résultats des instructions du p-code (calculs)
- Un segment de mémoire dans lequel s'effectue l'allocation et la désallocation d'objets.
- Une zone de stockage des méthodes contenant le p-code de chaque méthode et son environnement (tables des symboles,...)
- Un ensemble de registres 32 bits servant à mémoriser les différents états de la machine et les informations utiles à l'exécution de l'instruction présente dans le registre instruction bytecode en cours :
    - **s** : pointe dans la pile vers la première variable locale de la méthode en cours d'exécution.
    - **pc** : compteur ordinal indiquant l'adresse de l'instruction de p-code en cours d'exécution.
    - **optop** : sommet de pile des opérandes.
      frame



Deux objets Java seront instanciés dans la **machine virtuelle Java** de la manière suivante :

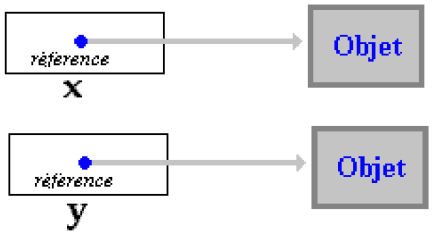


Attitude à rapprocher pour comparaison, à celle dont **Delphi** gère les objets dans une pile d'exécution de type LIFO et un tas :

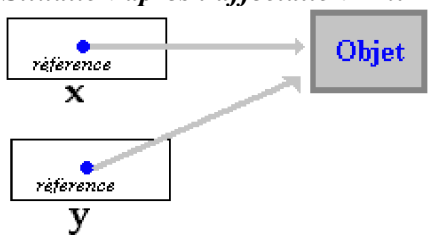


Attention à l'utilisation de l'affectation entre variables d'objets dans le modèle de représentation par référence. L'affectation x = y ne recopie pas le bloc objet de données de y dans celui de x, mais seulement la référence (l'adresse) de y dans la référence de x. Visualisons cette remarque importante :

Situation au départ, avant affectation



Situation après l'affectation " x = y "



En java, la désallocation étant automatique, le bloc de données objet qui était référencé par **y** avant l'affectation, n'est pas perdu, car le garbage collector se charge de restituer la mémoire libérée au **segment de mémoire** de la **machine virtuelle Java**.

Les constructeurs d'objets

Un constructeur est une **méthode spéciale** d'une classe dont la seule fonction est d'**instancier** un objet (créer le bloc de données). Comme en Delphi une **classe Java peut posséder plusieurs constructeur**s, il est possible de pratiquer des initialisations d'attributs dans un constructeur. Comme toutes les méthodes, un constructeur peut avoir ou ne pas avoir de paramètres formels.

- Si vous ne déclarez pas de constructeur spécifique pour une classe, **par défaut** Java attribue automatiquement un constructeur sans paramètres formels, portant le même nom que la classe. A la différence de Delphi où le nom du constructeur est quelconque, en Java le(ou les) **constructeur doit obligatoirement porter le même nom que la classe** (majuscules et minuscules comprises).
- Un constructeur d'objet d'une classe n'a d'intérêt que s'il est visible par tous les programmes qui veulent instancier des objets de cette classe, c'est pourquoi l'on mettra toujours le mot clef **public** devant la déclaration du constructeur.
- Un constructeur est une méthode spéciale dont la fonction est de créer des objets, dans son en-tête il n'a pas de type de retour et le mot clef **void** n'est pas non plus utilisé !

Soit une classe dénommée **Un** dans laquelle, comme nous l'avons fait jusqu'à présent nous n'indiquons aucun constructeur spécifique :

```
class Un
 { int a;
 }
```

Automatiquement Java attribue un constructeur public à cette classe **public** Un ( ). C'est comme si Java avait introduit dans votre classe à votre insu , une nouvelle méthode dénommée «Un ».Cette méthode "cachée" n'a aucun paramètre et aucune instruction dans son corps. Ci-dessous un exemple de programme Java correct illustrant ce qui se passe :

```
class Un
 { public Un ( ) { }
   int a;
 }
```

Possibilités de définition des constructeurs :

- **Vous pouvez programmer et personnaliser vos propres constructeurs.**
- **Une classe Java peut contenir plusieurs constructeurs dont les en-têtes diffèrent uniquement par la liste des paramètres formels.**

Exemple de constructeur avec instructions :

Java	Explication
class Un { **public** Un ( ) { a = 100 } **int** a; }	Le constructeur **public** Un sert ici à initialiser à 100 la valeur de l'attribut "**int** a" de chaque objet qui sera instancié.

Exemple de constructeur avec paramètre :

Java	Explication
class Un { **public** Un (**int** b) { a = b; } **int** a; }	Le constructeur **public** Un sert ici à initialiser la valeur de l'attribut "**int** a" de chaque objet qui sera instancié. Le paramètre **int** b contient cette valeur.

Exemple avec plusieurs constructeurs :

Java	Explication
class Un { **public** Un (**int** b)	La classe Un possède 3 constructeurs servant à initialiser chacun d'une manière

{ a = b; } **public** Un ( ) { a = 100; } **public** Un (**float** b) { a = (**int**)b; } **int** a; }	différente le seul attribut **int** a.

Comparaison Delphi - Java pour la déclaration de constructeurs

Delphi	Java
Un = **class** a : integer; **public** **constructor** creer; **overload**; **constructor** creer (b:integer); **overload**; **constructor** creer (b:real); **overload**; **end**; **implementation** **constructor** **Un**.creer; **begin** a := 100 **end**; **constructor** **Un**.creer(b:integer); **begin** a := b **end**; **constructor** **Un**.creer(b:real); **begin** a := trunc(b) **end**;	**class** **Un** { **public** Un ( ) { a = 100; } **public** Un (**int** b) { a = b; } **public** Un (**float** b) { a = (**int**)b; } **int** a; }

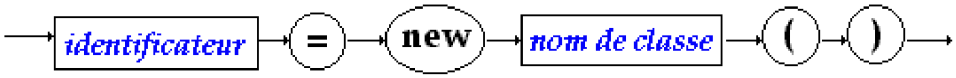
En Delphi un constructeur a un nom quelconque, tous les constructeurs peuvent avoir des noms différents ou le même nom comme en Java.

Utilisation du constructeur d'objet automatique (par défaut)

Le constructeur d'objet par défaut de toute classe Java comme nous l'avons signalé plus haut est une méthode spéciale sans paramètre, l'appel à cette méthode spéciale afin de construire un nouvel objet répond à une syntaxe spécifique par utilisation du mot clef **new**.

Syntaxe
Pour un constructeur sans paramètres formels, l'instruction d'**instanciation d'un nouvel objet** à

partir d'un identificateur de variable déclarée selon un type de classe, s'écrit ainsi :



Exemple : (deux façons équivalentes de créer un objet **x** de classe **Un**)

```
Un x ;
x = new Un( );      <=>      Un x = new Un( );
```

Cette instruction crée dans le segment de mémoire de la machine virtuelle Java, un nouvel objet de classe **Un** dont la référence (l'adresse) est mise dans la variable **x**

Dans l'exemple ci-dessous, nous utilisons le constructeur par défaut de la classe Un :

```
class Un
 { ...
 }

// la déclaration :
Un x , y ;
....
// la création :
x = new Un( );
y = new Un( );
```

Un programme de 2 classes, illustrant l'affectation de références :

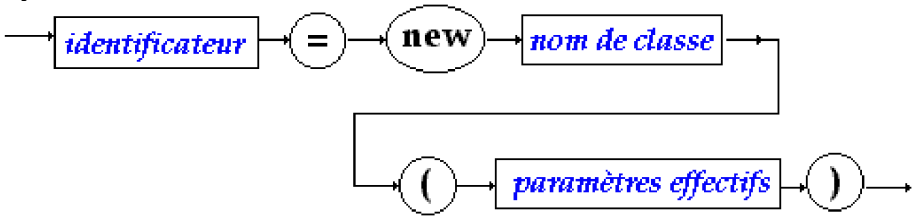
Java	Explication
class AppliClassesReferences { **public static void** main(String [] arg) { **Un** x,y ; x = **new Un**( ); y = **new Un**( ); System.out.println("x.a="+x.a); System.out.println("y.a="+y.a); y = x; x.a =12; System.out.println("x.a="+x.a); System.out.println("y.a="+y.a); } } **class Un** { **int** a=10; }	Ce programme Java contient deux classes : **class** AppliClassesReferences et **class** Un La classe AppliClassesReferences est une classe exécutable car elle contient la méthode main. C'est donc cette méthode qui agira dès l'exécution du programme.

Détaillons les instructions	**Que se passe-t-il à l'exécution ?**
Un x,y ; x = **new Un**( ); y = **new Un**( );	Instanciation de 2 objets différents x et y de type **Un**.
System.out.println("x.a="+x.a); System.out.println("y.a="+y.a);	*Affichage de :* x.a = 10 y.a = 10
y = x;	La référence de y est remplacée par celle de x dans la variable y (y pointe donc vers le même bloc que x).
x.a =12; System.out.println("x.a="+x.a); System.out.println("y.a="+y.a);	*On change la valeur de l'attribut a de x, et l'on demande d'afficher les attributs de x et de y* : x.a = 12 y.a = 12 **Comme y pointe vers x, y et x sont maintenant le même objet sous deux noms différents !**

Utilisation d'un constructeur d'objet personnalisé

L'utilisation d'un constructeur personnalisé d'une classe est semblable à celle du constructeur par défaut de la classe. La seule différence se trouve lors de l'instanciation : il faut fournir des paramètres effectifs lors de l'appel au constructeur.

Syntaxe



Exemple avec plusieurs constructeurs :

une classe Java	Des objets créés
class Un { **int** a ; **public** Un (**int** b) { a = b ; } **public** Un ( ) { a = 100 ; } **public** Un (**float** b) { a = (**int**)b ; }	**Un** obj1 = **newUn**( ); **Un** obj2 = **new Un**(15); **int** k = 14; **Un** obj3 = **new Un**(k); **Un** obj4 = **new Un**(3.25f); **float** r = -5.6; **Un** obj5 = **new Un**(r);

}	

Le mot clef this pour désigner un autre constructeur

Il est possible de dénommer dans les instructions d'une méthode de classe, un futur objet qui sera instancié plus tard. Le paramètre ou (mot clef) **this** est implicitement présent dans chaque objet instancié et il contient la référence à l'objet actuel. Il joue exactement le même rôle que le mot clef **self** en Delphi.

Java	Java équivalent
class Un { **public** Un ( ) { a = 100; } **int** a; }	**class Un** { **public** Un ( ) { **this**.a = 100; } **int** a; }

Dans le programme de droite le mot clef this fait référence à l'objet lui-même, ce qui dans ce cas est superflu puisque la variable **int** a est un champ de l'objet.

Montrons deux cas d'utilisation pratique de **this**

1° - Cas où l'objet est passé comme un paramètre dans une de ses méthodes :

Java	Explications
class Un { **public** Un ( ) { a = 100; } **public void** methode1(Un x) { System.out.println("champ a ="+x.a); } **public void** methode2(**int** b) { a += b; methode1(**this**); } **int** a; }	La methode1(Un x) reçoit un objet de type Exemple en paramètre et imprime son champ **int** a. La methode2(**int** b) reçoit un entier **int** b qu'elle additionne au champ **int** a de l'objet, puis elle appelle la méthode1 avec comme paramètre l'objet lui-même.

Comparaison Delphi - java sur cet exemple (similitude complète)

Delphi	Java
Un = **class** a : integer; **public** **constructor** creer; **procedure** methode1(x:Un); **procedure** methode2 (b:integer);	**class Un** { **public** Un ( ) { a = 100; } **public void** methode1(Un x) { System.out.println("champ a ="+x.a);

```
 end;

implementation

constructor Un.creer; begin
 a := 100
end;
procedure Un.methode1( x:Un );begin
 showmessage( 'champ a ='+inttostr(x.a) )
end;
procedure Un.methode2 ( b:integer );begin
 a := a+b;
 methode1(self)
end;
```

```
}
public void methode2( int b )
{  a += b;
   methode1(this);
}
 int a;
}
```

2° - Cas où le this sert à outrepasser le masquage de visibilité :

Java	Explications
`class Un` `{` `  int a;` `  public void methode1(float a)` `  {    a = this.a + 7 ;` `  }` `}`	La methode1(**float** a) possède un paramètre **float** a dont le nom masque le nom du champ **int** a. Si nous voulons malgré tout accéder au champ de l'objet, l'objet étant référencé par **this**, "**this**.a" est donc le champ **int** a de l'objet lui-même.

Comparaison Delphi - java sur ce second exemple (similitude complète aussi)

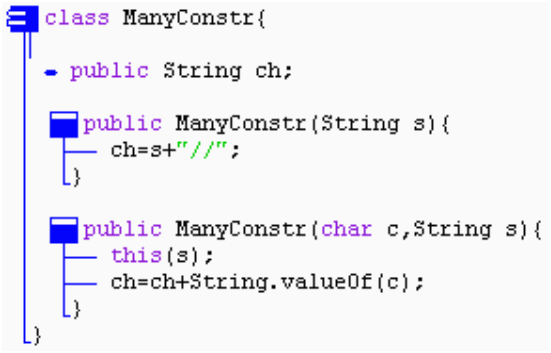
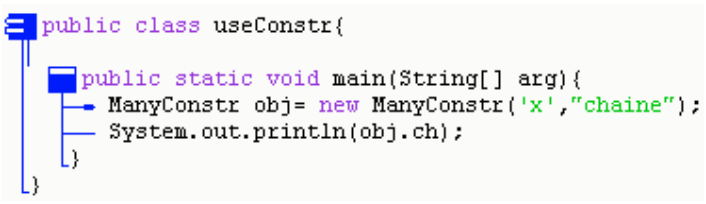
Delphi	Java
`Un = class` `    a : integer;` `  public` `   procedure methode( a:real );` `end;` `implementation` `procedure Un.methode( a:real );begin` `  a = self.a + 7 ;` `end;`	`class Un` `  {` `   int a;` `   public void methode(float a)` `   {    a = this.a + 7 ;` `   }` `  }`

Le this peut servir à désigner une autre surcharge de constructeur

Il est aussi possible d'utiliser le mot clef **this** en Java dans un constructeur pour désigner l'appel à un autre constructeur avec une autre signature. En effet comme tous les constructeurs portent le même nom, il a fallu trouver un moyen d'appeler un constructeur dans un autre constructeur, c'est le rôle du mot clef this que de jouer le rôle du nom standar du constructeur de la classe.

Lorsque le mot clef **this** est utilisé pour désigner une autre surcharge du constructeur en cours d'exécution, il doit **obligatoirement être la première instruction** du constructeur qui s'exécute (sous peine d'obtenir un message d'erreur à la compilation).

Exemple de classe à deux constructeurs :

Code Java	Explication
	La classe ManyConstr possède 2 constructeurs : Le premier : ManyConstr (String s) Le second : ManyConstr (char c, String s) Grâce à l'instruction **this**(s), le second constructeur appelle le premier sur la variable s, puis concatène le caractère **char** c au champ **String** ch.
	La méthode **main** instancie un objet de classe ManyConstr avec le second constructeur et affiche le contenu du champ **String** ch. De l'objet ; *Résultat de l'exécution :* **`chaine//x`**

Attributs et méthodes

Java2

Variables et méthodes

Nous examinons dans ce paragraphe comment Java utilise les variables et les méthodes à l'intérieur d'une classe. Il est possible de modifier des variables et des méthodes d'une classe ceci sera examiné plus loin.

En Java, les champs et les méthodes (ou **membres**) sont classés en deux catégories :

- **Variables et méthodes de classe**
- **Variables et méthodes d'instance**

Variables dans une classe en général

Rappelons qu'en Java, nous pouvons déclarer dans un bloc (for, try,...) de nouvelles variables à la condition qu'elles n'existent pas déjà dans le corps de la méthode où elles sont déclarées. Nous les dénommerons : **variables locales de méthode**.

Exemple de variables locales de méthode :

```
class Exemple
{
  void calcul ( int x, int y )
  {int a = 100;
    for ( int i = 1; i<10; i++ )
    {char carlu;
       System.out.print("Entrez un caractère : ");
       carlu = Readln.unchar( );
       int b =15;
       a =....
          .....
    }
  }
}
```

La définition **int** a = 100; est locale à la méthode en général

La définition **int** i = 1; est locale à la boucle **for**.

Les définitions **char** carlu et **int** b sont locales au corps de la boucle **for**.

Java ne connaît pas la notion de variable globale au sens habituel donné à cette dénomination, dans la mesure où toute variable ne peut être définie qu'à l'intérieur d'une classe, ou d'une méthode inclue dans une classe. Donc mis à part les **variables locales de méthode** définies dans une méthode, Java reconnaît une autre catégorie de variables, ***les variables définies dans une classe mais pas à l'intérieur d'une méthode spécifique***. Nous les dénommerons : **attributs de classes** parce que ces variables peuvent être de deux catégories.

Exemple de attributs de classe :

class AppliVariableClasse { **float** r ; **void** calcul (**int** x, **int** y) { .... } **int** x =100; **int** valeur (**char** x) { ..... } **long** y; }	Les variables float r , long y et int x sont des attributs de classe (ici en fait plus précisément, des variables d'instance). La position de la déclaration de ces variables n'a aucune importance. Elles sont visibles dans tout le bloc classe (c'est à dire visibles par toutes les méthodes de la classe). **Conseil : regroupez les variables de classe au début de la classe afin de mieux les gérer.**

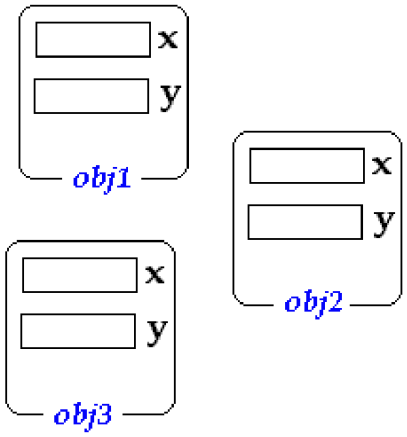
Les attributs de classe peuvent être soit de la catégorie des **variables de classe**, soit de la catégorie des **variables d'instance**.

Variables et méthodes d'instance

Java se comporte comme un langage orienté objet classique vis à vis de ses variables et de ses méthodes. A chaque instanciation d'un nouvel objet d'une classe donnée, la machine virtuelle Java enregistre le p-code des méthodes de la classe dans la **zone de stockage** des méthodes,elle alloue dans le **segment de mémoire** autant d'emplacements mémoire pour les variables que d'objet créés. Java dénomme cette catégorie **les variables et les méthodes d'instance**.

une classeJava	Instanciation de 3 objets
class AppliInstance { **int** x ; **int** y ; }	AppliInstance obj1 = **new** AppliInstance( ); AppliInstance obj2 = **new**AppliInstance( ); AppliInstance obj3 = **new**AppliInstance( );

Voici une image du segment de mémoire associé à ces 3 objets :



Un programme Java à 2 classes illustrant l'exemple précédent :

Programme Java exécutable

```
class AppliInstance
{  int x = -58 ;
   int y = 20 ;
}
class Utilise
{ public static void main(String [ ] arg) {
     AppliInstance  obj1 = new AppliInstance( );
     AppliInstance  obj2 = new AppliInstance( );
     AppliInstance  obj3 = new AppliInstance( );
     System.out.println( "obj1.x = " + obj1.x );
  }
}
```

Variables et méthodes de classe - static

Variable de classe

On identifie une variable ou une méthode de classe en précédant sa déclaration du mot clef **static**. Nous avons déjà pris la majorité de nos exemples simples avec de tels composants.

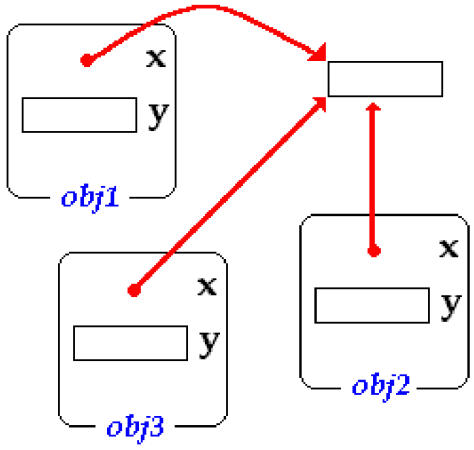
Voici deux déclarations de variables de classe :

```
static int x ;
static int a = 5;
```

Une variable de classe est accessible comme une variable d'instance (selon sa visibilité), mais aussi **sans avoir à instancier un objet de la classe**, uniquement en référençant la variable par le nom de la classe dans la notation de chemin uniforme d'objet.

une classeJava	Instanciation de 3 objets
class AppliInstance { **static int** x ; **int** y ; }	AppliInstance obj1 = **new** AppliInstance ( ); AppliInstance obj2 = **new** AppliInstance ( ); AppliInstance obj3 = **new** AppliInstance ( );

Voici une image du segment de mémoire associé à ces 3 objets :

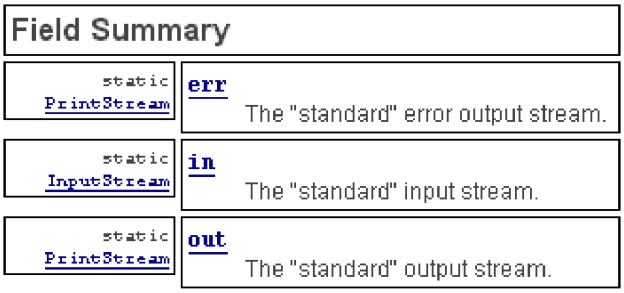


Exemple de variables de classe :

class ApplistaticVar { **static int** x =15 ; } **class** UtiliseApplistaticVar { **int** a ; **void** f( ) { a = ApplistaticVar.x ; ..... } }	La définition "**static int** x =15 ;" crée une variable de la classe ApplistaticVar, nommée x. L'instruction "a = ApplistaticVar.x ;" utilise la variable x comme variable de classe ApplistaticVar sans avoir instancié un objet de cette classe.

Nous utilisons sans le savoir depuis le début de ce cours, une variable de classe sans jamais instancier un quelconque objet de la classe. Dans l'instruction << **System.out.println ("Bonjour");** >>, la classe **System** possède une variable (un champ)de classe **out** qui est elle-même un objet de classe dérivant de la classe **FilterOutputStream**, nous n'avons jamais instancié d'objet de cette classe **System**.

Les champs de la classe System :



Field Summary	
static PrintStream	**err** The "standard" error output stream.
static InputStream	**in** The "standard" input stream.
static PrintStream	**out** The "standard" output stream.

Notons que les champs **err** et **in** sont aussi des variables de classe (précédées par le mot **static**).

Méthode de classe

Une méthode de classe est une méthode dont l'implémentation est la même pour tous les objets de la classe, en fait la différence avec une méthode d'instance a lieu sur la catégorie des variables sur lesquelles ces méthodes agissent.

De par leur définition les méthodes de classe ne peuvent travailler qu'avec des variables de classe, alors que les méthodes d'instances peuvent utiliser les deux catégories de variables.

Un programme correct illustrant le discours :

Java	Explications
class Exemple { **static int** x ; **int** y ; **void** f1(**int** a) { x = a; y = a; } **static void** g1(int a) { x = a; } } **class** Utilise { **public static void** main(String [] arg) { Exemple obj = **new** Exemple( ); obj.f1(10); System.out.println("<f1(10)>obj.x="+obj.x); obj.g1(50); System.out.println("<g1(50)>obj.x="+obj.x); } }	**void** f1(**int** a) { x = a; *//accès à la variable de classe* y = a ; *//accès à la variable d'instance* } **static void** g1(**int** a) { x = a; *//accès à la variable de classe* y = a ; *//engendrerait une erreur de compilation : accès à une variable non static interdit !* } La méthode f1 accède à toutes les variables de la classe Exemple, la méthode g1 n'accède qu'aux variables de classe (**static**). ***Après exécution on obtient :*** <f1(10)>obj.x = 10 <g1(50)>obj.x = 50

Résumons ci-dessous ce qu'il faut connaître pour bien utiliser ces outils.

Bilan pratique et utile sur les membres de classe, en 5 remarques

1) - *Les méthodes et les variables de classe sont **précédées obligatoirement** du mot clef **static**. Elles jouent un rôle **semblable** à celui qui est attribué aux variables et aux sous-routines globales dans un langage impératif classique.*

Java	Explications
class Exemple1 { **int** a = 5; **static int** b = 19; **void** m1( ){...} **static void** m2( ) {...} }	La variable a dans **int** a = 5; est une variable d'instance. La variable b dans **static int** b = 19; est une variable de classe. La méthode m2 dans **static void** m2( ) {...} est une méthode de classe.

2) - *Pour utiliser une variable **x1** ou une méthode **meth1** de la classe **Classe1**, il suffit de d'écrire **Classe1.x1** ou bien **Classe1.meth1**.*

Java	Explications
class Exemple2 { **static int** b = 19; **static void** m2( ) {...} } **class** UtiliseExemple { Exemple2.b = 53; Exemple2.m2( ); ... }	Dans la classe Exemple2 b est une variable de classe, m2 une méthode de classe. La classe UtiliseExemple fait appel à la méthode m2 directement avec le nom de la classe, il en est de même avec le champ b de la classe Exemple2.

3) - *Une variable de classe (précédée du mot clef **static**) est **partagée par tous les objets** de la même classe.*

Java	Explications
class AppliStatic { **static int** x = -58 ; **int** y = 20 ; ... } **class** Utilise { **public static void** main(String [] arg) { AppliStatic obj1 = **new** AppliStatic( ); AppliStatic obj2 = **new** AppliStatic( ); AppliStatic obj3 = **new** AppliStatic( ); obj1.y = 100; obj1.x = 101; System.out.println("obj1.x="+obj1.x); System.out.println("obj1.y="+obj1.y); System.out.println("obj2.x="+obj2.x); System.out.println("obj2.y="+obj2.y); System.out.println("obj3.x="+obj3.x); System.out.println("obj3.y="+obj3.y); AppliStatic.x = 99; System.out.println(AppliStatic.x="+obj1.x); } }	Dans la classe AppliStatic **x** est une variable de classe, et **y** une variable d'instance. La classe Utilise crée 3 objets (obj1, obj2, obj3) de classe AppliStatic. L'instruction obj1.**y** = 100; est un accès au champ **y** de l'instance obj1. Ce n'est que le champ **x** de cet objet qui est modifié, les champs **x** des objets obj2 et obj3 restent inchangés Il y a deux manières d'accéder à la variable static **x** : soit comme un champ de l'objet (accès semblable à celui de **y**) : obj1.**x** = 101; soit comme une variable de classe proprement dite : AppliStatic.**x** = 99; Dans les deux cas cette variable **x** est modifiée globalement et donc tous les champs **x** des 2 autres objets, obj2 et obj3 prennent la nouvelle valeur.

Au début lors de la création des 3 objets chacun des champs x vaut -58 et des champs y vaut 20, l'affichage par System.out.println(...) donne les résultats suivants qui démontrent le partage de la variable x par tous les objets.

Après exécution :
obj1.x = 101
obj1.y = 100
obj2.x = 101
obj2.y = 20
obj3.x = 101
obj3.y = 20
<AppliStatic>obj1.x = 99

4) - *Une méthode de classe (précédée du mot clef **static**) **ne peut utiliser que des variables de classe** (précédées du mot clef **static**) et jamais des variables d'instance.Une méthode d'instance peut accéder aux deux catégories de variables.*

5) - *Une méthode de classe (précédée du mot clef **static**) **ne peut appeller** (invoquer) **que des méthodes de classe** (précédées du mot clef **static**).*

Java	Explications
class AppliStatic { **static int** x = -58 ; **int** y = 20 ; **void** f1(**int** a) { AppliStatic.x = a; y = 6 ; } } **class** Utilise { **static void** f2(int a) { AppliStatic.x = a; } **public static void** main(String [] arg) { AppliStatic obj1 = new AppliStatic( ); AppliStatic obj2 = new AppliStatic( ); AppliStatic obj3 = new AppliStatic( ); obj1.y = 100; obj1.x = 101; AppliStatic.x = 99; f2(101); obj1.f1(102); } }	Nous reprenons l'exemple précédent en ajoutant à la classe AppliStatic une méthode interne f1 : **void** f1(**int** a) { AppliStatic.x = a; y = 6 ; } Cette méthode accède à la variable de classe comme un champ d'objet. Nous rajoutons à la classe Utilise, une méthode static (méthode de classe) notée f2: **static void** f2(int a) { AppliStatic.x = a; } Cette méthode accède elle aussi à la variable de classe parce qu'c'est une méthode static. Nous avons donc **quatre** manières d'accéder à la variable static **x**, : soit comme un champ de l'objet (accès semblable à celui de **y**) : obj1.**x** = 101; soit comme une variable de classe proprement dite : AppliStatic.**x** = 99; soit par une méthode d'instance sur son champ : obj1.f1(102); soit par une méthode static (de classe) : f2(101);

Comme la méthode main est static, elle peut invoquer la méthode f2 qui est aussi statique.

Au paragraphe précédent, nous avons indiqué que Java ne connaissait pas la notion de variable globale stricto sensu, mais en fait une variable **static peut jouer le rôle d'un variable globale pour un ensemble d'objets** instanciés à partir de la même classe.

Surcharge et polymorphisme

Vocabulaire :
Le polymorphisme est la capacité d'une entité à posséder plusieurs formes. En informatique ce vocable s'applique aux objets et aussi aux méthodes selon leur degré d'adaptabilité, nous distinguons alors deux dénominations :

- A - le polymorphisme statique ou **la surcharge de méthode**

- B- le polymorphisme dynamique ou **la redéfinition de méthode** ou encore la **surcharge héritée**.

A - La surcharge de méthode (polymorphisme statique)

C'est une fonctionnalité classique des langages très évolués et en particulier des langages orientés objet; elle consiste dans le fait qu'une classe peut disposer de **plusieurs méthodes ayant le même nom**, mais avec des paramètres formels différents ou éventuellement un type de retour différent. On appelle **signature** de la méthode l'en-tête de la méthode avec ses paramètres formels. Nous avons déjà utilisé cette fonctionnalité précédement dans le paragraphe sur les constructeurs, où la classe **Un** disposait de trois constructeurs surchargés :

```
class Un
 {
  int a;
   public Un ( )
   { a = 100;   }

   public Un (int b )
   { a = b;   }

   public Un (float b )
   { a = (int)b;   }
}
```

Mais cette surcharge est possible aussi pour n'importe quelle méthode de la classe autre que le constructeur. Le compilateur n'éprouve aucune difficulté lorsqu'il rencontre un appel à l'une des versions surchargée d'une méthode, il cherche dans la déclaration de toutes les surcharges celle dont la **signature** (la déclaration des paramètres formels) coïncide avec les paramètres effectifs de l'appel.

Programme Java exécutable	Explications
class Un { int a; **public** Un (**int** b) { a = b; }	La méthode **f** de la classe **Un** est surchargée trois fois : **void** f ( )

```
  void f ( )
  { a *=10;  }
  void f ( int x )
  { a +=10*x; }
  int f ( int x, char y )
  { a = x+(int)y;
    return a;  }
}
class AppliSurcharge
{
   public static void main(String [ ] arg) {
   Un obj = new Un(15);
   System.out.println("<création> a ="+obj.a);
   obj.f( );
   System.out.println("<obj.f()> a ="+obj.a);
   obj.f(2);
   System.out.println("<obj.f()> a ="+obj.a);
   obj.f(50,'a');
   System.out.println("<obj.f()> a ="+obj.a);
   }
}
```

```
  { a *=10;  }

void f ( int x )
  { a +=10*x; }

int f ( int x, char y )
  { a = x+(int)y;
    return a;  }
```

La méthode f de la classe Un peut donc être appelée par un objet instancié de cette classe sous l'une quelconque des trois formes :

obj.f( ); pas de paramètre => choix : **void** f ( )

obj.f(2); paramètre **int** => choix : **void** f (**int** x)

obj.f(50,'a'); deux paramètres, un int un char => choix : **int** f (**int** x, **char** y)

Comparaison Delphi - java sur la surcharge :

Delphi	Java
Un = **class** a : integer; **public** **constructor** methode(b : integer); **procedure** f;**overload**; **procedure** f(x:integer);**overload**; **function** f(x:integer;y:char):integer;**overload**; **end**; **implementation** **constructor** Un.methode(b : integer); **begin** a:=b **end**; **procedure** Un.f; **begin** a:=a*10; **end**; **procedure** Un.f(x:integer); **begin** a:=a+10*x **end**; **function** Un.f(x:integer;y:char):integer; **begin** a:=x+ord(y); result:= a **end**; **procedure** LancerMain;	**class** Un { int a; **public** Un (**int** b) { a = b; } **void** f ( ) { a *=10; } **void** f (**int** x) { a +=10*x; } **int** f (**int** x, **char** y) { a = x+(**int**)y; **return** a; } } **class** AppliSurcharge { **public static void** main(String [ ] arg) { Un obj = **new** Un(15); System.out.println("<création> a ="+obj.a); obj.f( ); System.out.println("<obj.f()> a ="+obj.a); obj.f(2); System.out.println("<obj.f()> a ="+obj.a); obj.f(50,'a'); System.out.println("<obj.f()> a ="+obj.a);

```
var obj:Un;
begin
 obj:=Un.methode(15);
 obj.f;
 Memo1.Lines.Add('obj.f='+inttostr(obj.a));
 obj.f(2);
 Memo1.Lines.Add('obj.f(2)='+inttostr(obj.a));
 obj.f(50,'a');
Memo1.Lines.Add('obj.f(50,"a")='+inttostr(obj.a));
end;
```

```
  }
}
```

B - La redéfinition de méthode (polymorphisme dynamique)

C'est une fonctionnalité spécifique aux langages orientés objet. Elle est mise en oeuvre lors de l'héritage d'une classe mère vers une classe fille dans le cas d'une méthode ayant la même signature dans les deux classes. Dans ce cas les actions dûes à l'appel de la méthode, dépendent du code inhérent à chaque version de la méthode (celle de la classe mère, ou bien celle de la classe fille). Ces actions peuvent être différentes. En java aucun mot clef n'est nécessaire ni pour la surcharge ni pour la redéfinition, c'est le compilateur qui analyse la syntaxe afin de de se rendre compte en fonction des signatures s'il s'agit de redéfinition ou de surcharge. Attention il n'en va pas de même en Delphi, plus verbeux mais plus explicite pour le programmeur, qui nécessite des mots clefs comme virtual, dynamic override et overload.

Dans l'exemple ci-dessous la classe ClasseFille qui hérite de la classe ClasseMere, redéfinit la méthode **f** de sa classe mère :

Comparaison redéfinition Delphi et Java :

Delphi

```
type
 ClasseMere = class
   x : integer;
   procedure f (a:integer);virtual;//autorisation
   procedure g(a,b:integer);
 end;

ClasseFille = class ( ClasseMere )
  y : integer;
   procedure f (a:integer);override;//redéfinition
   procedure g1(a,b:integer);
 end;

implementation

procedure ClasseMere.f (a:integer); begin...
end;
procedure ClasseMere.g(a,b:integer); begin...
end;
procedure ClasseFille.f (a:integer); begin...
```

Java

```
class ClasseMere
{
  int x = 10;

void f ( int a)
  { x +=a; }
  void g ( int a, int b)
  { x +=a*b; }
}


class ClasseFille extends ClasseMere
{
   int y = 20;
  void f ( int a) //redéfinition
  { x +=a; }
  void g1 (int a, int b) //nouvelle méthode
  { ..... }
}
```

end; **procedure** ClasseFille.g1(a,b:integer); **begin...** **end**;	

Comme delphi, Java peut combiner la surcharge et la redéfinition sur une même méthode, c'est pourquoi nous pouvons parler de **surcharge héritée** :

Java

```
class ClasseMere
{
 int x = 10;

        void f ( int a)
        { x +=a; }
        void g ( int a, int b)
        { x +=a*b; }
       }

       class ClasseFille extends ClasseMere
       {
         int y = 20;
        void f ( int a) //redéfinition
        { x +=a; }
        void g (char b) //surcharge et redéfinition de g
        { ..... }
       }
```

C'est le compilateur Java qui fait tout le travail. Prenons un objet obj de classe Classe1, lorsque le compilateur Java trouve une instruction du genre "obj.**method1**(paramètres effectifs);", sa démarche d'analyse est semblable à celle du compilateur Delphi, il cherche dans l'ordre suivant :

- Y-a-t-il dans Classe1, une méthode qui se nomme **method1** ayant une signature identique aux paramètres effectifs ?
- si oui c'est la méthode ayant cette signature qui est appelée,
- si non le compilateur remonte dans la hierarchie des classes mères de Classe1 en posant la même question récursivement jusqu'à ce qu'il termine sur la classe Object.
- Si aucune méthode ayant cette signature n'est trouvée il signale une erreur.

Soit à partir de l'exemple précédent les instructions suivantes :
ClasseFille obj = **new** ClasseFille( );
obj.**g**(-3,8);
obj.**g**('h');

Le compilateur Java applique la démarche d'analyse décrite, à l'instruction "obj.**g**(-3,8);". Ne trouvant pas dans ClasseFille de méthode ayant la bonne signature (signature = deux entiers) , le compilateur remonte dans la classe mère ClasseMere et trouve une méthode " **void** g (**int** a, **int** b) " de la classe ClasseMere ayant la bonne signature (signature = deux entiers), il procède alors à l'appel de cette méthode sur les paramètres effectifs (-3,8).

Dans le cas de l'instruction obj.**g**('h'); , le compilateur trouve immédiatement dans ClasseFille la méthode " **void** g (**char** b) " ayant la bonne signature, c'est donc elle qui est appelée sur le paramètre effectif 'h'.

Résumé pratique sur le polymorphisme en Java

La **surcharge** (polymorphisme statique) consiste à proposer différentes signatures de la même méthode.
La **redéfinition** (polymorphisme dynamique) ne se produit que dans l'héritage d'une classe par redéfinition de la méthode mère avec une méthode fille (ayant ou n'ayant pas la même signature).

Le mot clef super

Nous venons de voir que le compilateur s'arrête dès qu'il trouve une méthode ayant la bonne signature dans la hiérarchie des classes, il est des cas où nous voudrions accéder à une méthode de la classe mère alors que celle-ci est redéfinie dans la classe fille. C'est un problème analogue à l'utilisation du this lors du masquage d'un attribut. Il existe un mot clef qui permet d'accéder à la classe mère (classe immédiatement au dessus): le mot **super**.

On parle aussi de **super-classe** au lieu de classe mère en Java. Ce mot clef **super** référence la classe mère et à travers lui, il est possible d'accéder à tous les champs et à toutes les méthodes de la super-classe (classe mère). Ce mot clef est très semblable au mot clef **inherited** de Delphi qui joue le même rôle uniquement sur les méthodes.

Exemple :

```
class ClasseMere
{
  int x = 10;

  void g ( int a, int b)
  { x +=a*b; }
}

class ClasseFille extends ClasseMere
{
```

```
    int x = 20; //masque le champ x de la classe mère

   void g (char b) //surcharge et redéfinition de g
   {  super.x = 21; //accès au champ x de la classe mère
      super.g(-8,9); //accès à la méthode g de la classe mère
    }
}
```

Le mot clef super peut en Java être utilisé seul ou avec des paramètres comme un appel au constructeur de la classe mère.

Exemple :

```
class ClasseMere
{
  public ClasseMere ( ) {
  ... }
  public ClasseMere (int a ) {
  ... }

}

class ClasseFille extends ClasseMere
{
   public ClasseFille ( ) {
   super ( ); //appel au 1er constructeur de ClasseMere
   super ( 34 ); //appel au 2nd constructeur de ClasseMere
    ... }
  public ClasseFille ( char k, int x) {
   super ( x ); //appel au 2nd constructeur de ClasseMere
    ... }
}
```

Modification de visibilité

Terminons ce chapitre par les classiques modificateurs de visibilité des variables et des méthodes dans les langages orientés objets, dont Java dispose :

Modification de visibilité (modularité public-privé)

par défaut **(aucun mot clef)**	les variables et les méthodes d'une classe non précédées d'un mot clef sont visibles par toutes les classes inclues dans le module seulement.
public	les variables et les méthodes d'une classe précédées du mot clef **public** sont visibles par toutes les classes de tous les modules.
private	les variables et les méthodes d'une classe précédées du mot clef **private** ne sont visibles que dans la classe seulement.

protected	les variables et les méthodes d'une classe précédées du mot clef **protected** sont visibles par toutes les classes inclues dans le module, et par les classes dérivées de cette classe.

Ces attributs de visibilité sont identiques à ceux de Delphi.

Les interfaces

Java2

Introduction

- Les interfaces ressemblent aux classes abstraites sur un seul point : elles contiennent des membres **expliquant certains comportements sans les implémenter**.

- Les classes abstraites et les interfaces se différencient principalement par le fait qu'**une classe peut implémenter un nombre quelconque d'interfaces**, alors qu'une classe abstraite ne peut hériter que d'**une seule classe** abstraite ou non.

Vocabulaire et concepts :

- Une interface est un contrat, elle peut contenir des **propriétés**, des **méthodes** et des **événements** mais **ne** doit contenir **aucun champ** ou **attribut**.
- Une interface **ne** peut **pas** contenir des méthodes déjà implémentées.
- Une interface doit contenir des méthodes **non** implémentées.
- Une interface est héritable.
- On peut construire une hiérarchie d'interfaces.
- Pour pouvoir construire un objet à partir d'une interface, il faut définir une classe non abstraite implémentant **toutes** les méthodes de l'interface.

Une classe peut implémenter plusieurs interfaces. Dans ce cas nous avons une excellente alternative à l'héritage multiple.

Lorsque l'on crée une interface, on fournit un ensemble de définitions et de comportements qui ne devraient plus être modifiés. Cette attitude de constance dans les définitions, protège les applications écrites pour utiliser cette interface.

Les variables de types interface respectent les mêmes règles de **transtypage** que les variables de types classe.

Les **objets** de type classe **clA** peuvent être transtypés et **référencés** par des variables d'interface **IntfA** dans la mesure où la classe **clA implémente l'interface IntfA**. (cf. polymorphisme d'objet)

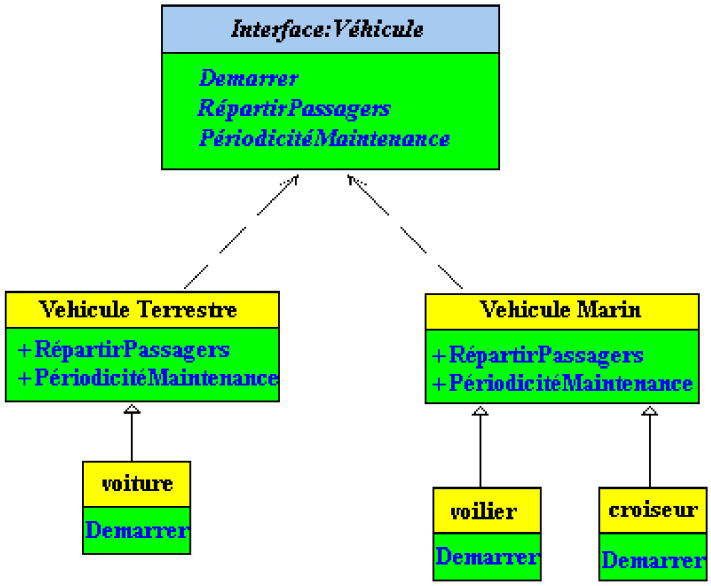
Si vous voulez utiliser la notion d'interface pour fournir un polymorphisme à une famille de classes, elles doivent toutes implémenter cette interface, comme dans l'exemple ci-dessous.

Exemple :

l'interface ***Véhicule*** définissant 3 méthodes (abstraites) **Démarrer**, **RépartirPassagers** de répartition des passagers à bord du véhicule (fonction de la forme, du nombre de places, du personnel chargé de s'occuper de faire fonctionner le véhicule...), et **PériodicitéMaintenance** renvoyant la périodicité de la maintenance obligatoire du véhicule (fonction du nombre de kms ou miles parcourus, du nombre d'heures d'activités,...)

Soit l'interface ***Véhicule*** définissant ces 3 méthodes :

Soient les deux classes **Véhicule terrestre** et **Véhicule marin**, qui implémentent partiellement chacune l'interface ***Véhicule*** , ainsi que trois classes **voiture**, **voilier** et **croiseur** héritant de ces deux classes :



- Les trois méthodes de l'interface ***Véhicule*** sont abstraites et publics par définition.

- Les classes **Véhicule terrestre** et **Véhicule marin** sont abstraites, car la méthode

abstraite **Démarrer** de l'interface ***Véhicule*** n'est pas implémentée elle reste comme "modèle" aux futurs classes. C'est dans les classes **voiture**, **voilier** et **croiseur** que l'on implémente le comportement précis du genre de démarrage.

Dans cette vision de la hiérarchie on a supposé que les classes abstraites **Véhicule terrestre** et **Véhicule marin** savent comment répartir leurs éventuels passagers et quand effectuer une maintenance du véhicule.

Les classes **voiture**, **voilier** et **croiseur** , n'ont plus qu'à implémenter chacune son propre comportement de démarrage.

Syntaxe de l'interface en Delphi et en Java (C# est semblable à Java) :

Delphi	Java
Vehicule = Interface **procedure** Demarrer; **procedure** RépartirPassagers; **procedure** PériodicitéMaintenance; **end;**	**Interface** Vehicule { **void** Demarrer( ); **void** RépartirPassagers( ); **void** PériodicitéMaintenance( ); }

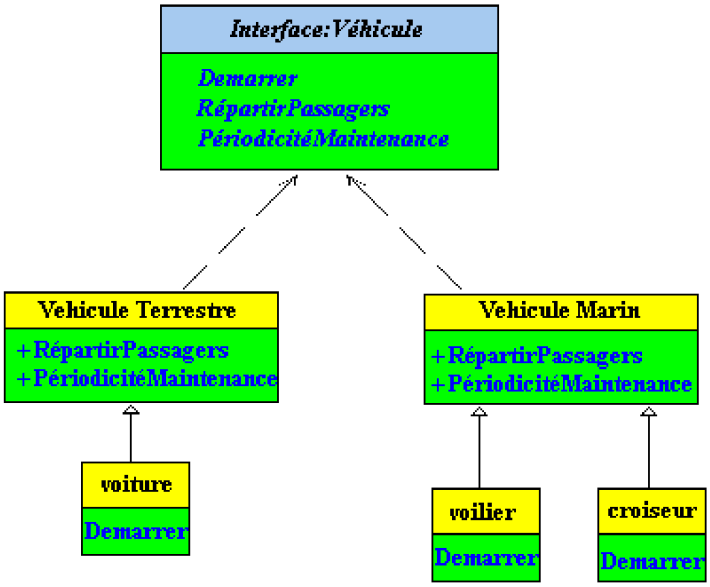
Utilisation pratique des interfaces

Quelques conseils prodigués par des développeurs professionnels (microsoft, Borland) :

- Les interfaces bien conçues sont plutôt petites et indépendantes les unes des autres.
- Un trop grand nombre de fonctions rend l'interface peu maniable.
- Si une modification s'avère nécessaire, une nouvelle interface doit être créée.
- La décision de créer une fonctionnalité en tant qu'interface ou en tant que classe abstraite peut parfois s'avérer difficile.
- Vous risquerez moins de faire fausse route en concevant des interfaces qu'en créant des arborescences d'héritage très fournies.
- Si vous projetez de créer plusieurs versions de votre composant, optez pour une classe abstraite.
- Si la fonctionnalité que vous créez peut être utile à de nombreux objets différents, faites appel à une interface.
- Si vous créez des fonctionnalités sous la forme de petits morceaux concis, faites appel aux interfaces.
- L'utilisation d'interfaces permet d'envisager une conception qui sépare la manière d'utiliser une classe de la manière dont elle est implémentée.

- Deux classes peuvent partager la même interface sans descendre nécessairement de la même classe de base.

Exemple de hiérarchie à partir d'une interface :



Dans cet exemple :

Les méthodes RépartirPassagers, PériodicitéMaintenance et Demarrer sont implantées soit comme des méthodes à liaison dynamique afin de laisser la possibilité pour des classes enfants de surcharger ces méthodes.

Soit l'écriture en Java de cet l'exemple :

```
interface IVehicule{
      void Demarrer( );
      void RépartirPassager( );
      void PériodicitéMaintenance( );
  }

abstract class Terrestre  implements IVehicule {
      public void RépartirPassager( ){.....};
      public void PériodicitéMaintenance( ){.....};
 }

class Voiture extends Terrestre {
      public void Demarrer( ){.....};
 }

abstract class Marin implements IVehicule {
```

```
        public void RépartirPassager( ){.....};
        public void PériodicitéMaintenance( ){.....};
}


class Voilier extends Marin {
        public void Demarrer( ){.....};
}
 class Croiseur extends Marin {
        public void Demarrer( ){.....};
}
```

Java2 à la fenêtre
avec Awt

IHM avec Java

Java, comme tout langage moderne, permet de créer des applications qui ressemblent à l'interface du système d'exploitation. Cette assurance d'ergonomie et d'interactivité avec l'utilisateur est le minimum qu'un utilisateur demande à une application. Les interfaces homme-machine (dénommées IHM) font intervenir de nos jours des éléments que l'on retrouve dans la majorité des systèmes d'exploitation : les fenêtres, les menus déroulants, les boutons, etc...

Ce chapitre traite en résumé, mais en posant toutes les bases, de l'aptitude de Java à élaborer une IHM. Nous regroupons sous le vocable d'IHM Java, les applications disposant d'une interface graphique et les applets que nous verrons plus loin.

Le package AWT

C'est pour construire de telles IHM que le package AWT (Abstract Window Toolkit) est inclu dans toutes les versions de Java. Ce package est la base des extensions ultérieures comme **Swing**, mais est le seul qui fonctionne sur toutes les générations de navigateurs.

Les classes contenues dans AWT dérivent (héritent) toutes de la classe **Component**, nous allons étudier quelques classes minimales pour construire une IHM standard.

Les classes Conteneurs

Ces classes sont essentielles pour la construction d'IHM Java elles dérivent de la classe **java.awt.Container**, elles permettent d'intégrer d'autres objets visuels et de les organiser à l'écran.

Hiérarchie de la classe* Container *:

```
java.lang.Object
 |
 +--java.awt.Component
      |
      +--java.awt.Container
```

Voici la liste extraite du JDK des sous-classes de la classe **Container** autres que **Swing** : **Panel, ScrollPane, Window.**

Les principales classes conteneurs :

Classe	Fonction
+--java.awt.Container | +--**java.awt.Window**	Crée des rectangles simples sans cadre, sans menu, sans titre, mais ne permet pas de créer directement une fenêtre Windows classique.
+--java.awt.Container | +--**java.awt.Panel**	Crée une surface sans bordure, capable de contenir d'autres éléments : boutons, panel etc...
+--java.awt.Container | +--**java.awt.ScrollPane**	Crée une barre de défilement horizontale et/ou une barre de défilement verticale.

Les classes héritées des classes conteneurs :

Classe	Fonction
java.awt.Window | +--**java.awt.Frame**	Crée des fenêtres avec bordure, pouvant intégrer des menus, avec un titre, etc...comme toute fenêtre Windows classique. C'est le conteneur de base de toute application graphique.
java.awt.Window | +--**java.awt.Dialog**	Crée une fenêtre de dialogue avec l'utilisateur, avec une bordure, un titre et un bouton-icône de fermeture.

Une première fenêtre construite à partir d'un objet de classe Frame; une fenêtre est donc un objet, on pourra donc créer autant de fenêtres que nécessaire, il suffira à chaque fois d'instancier un objet de la classe Frame.

Quelques méthodes de la classe Frame, utiles au départ :

Méthodes	Fonction
public void setSize(**int** width, **int** height)	retaille la largeur (width) et la hauteur (height) de la fenêtre.
public void setBounds(**int** x, **int** y, **int** width, **int** height)	retaille la largeur (width) et la hauteur (height) de la fenêtre et la positionne en x,y sur l'écran.

public Frame(**String** title) **public** Frame( )	Les deux constructeurs d'objets Frame, celui qui possède un paramètre String écrit la chaîne dans la barre de titre de la fenêtre.
public void setVisible(boolean b)	Change l'état de la fenêtre en mode **visible** ou **invisible** selon la valeur de b.
public void hide( )	Change l'état de la fenêtre en mode **invisible**.
Différentes surcharges de la méthode add : **public Component add(Component** comp) etc...	Permettent d'ajouter un composant à l'intérieur de la fenêtre.

Une Frame lorsque son constructeur la crée est en mode invisible, il faut donc la rendre visible, c'est le rôle de la méthode **setVisible (true)** que vous devez appeler afin d'afficher la fenêtre sur l'écran :

Programme Java

```
import java.awt.*;
class AppliWindow
{
  public static void main(String [ ] arg) {
    Frame fen = new Frame ("Bonjour" );
    fen. setVisible ( true );
   }
}
```

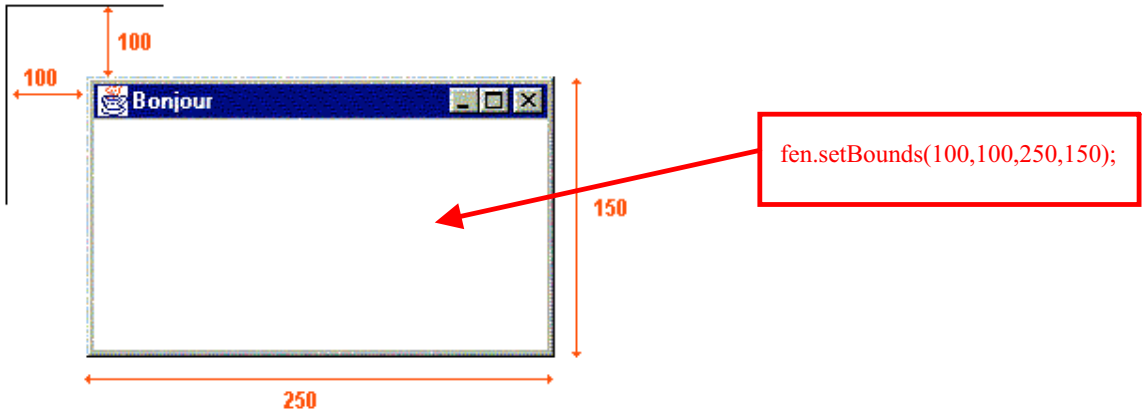
Ci-dessous la fenêtre affichée par le programme précédent :

Cette fenêtre est trop petite, retaillons-la avec la méthode setBounds :

Programme Java

```
import java.awt.*;
class AppliWindow
{
  public static void main(String [ ] arg) {
    Frame fen = new Frame ("Bonjour" );
    fen.setBounds(100,100,250,150);
    fen. setVisible ( true );
   }
}
```

Ci-dessous la fenêtre affichée par le programme précédent :



Pour l'instant nos fenêtres sont repositionnables, retaillables, mais elles ne contiennent rien, comme ce sont des objets conteneurs, il est possible en particulier, d'y inclure des composants.

Il est possible d'afficher des fenêtres dites de dialogue de la classe Dialog, dépendant d'une Frame. Elles sont très semblables aux Frame (barre de titre, cadre,...) mais ne disposent que d'un bouton icône de fermeture dans leur titre :

une fenêtre de classe Dialog :

De telles fenêtres doivent être obligatoirement rattachées lors de la construction à un parent qui sera une Frame ou une autre boîte de classe Dialog, le constructeur de la classe Dialog contient plusieurs surcharges dont la suivante :

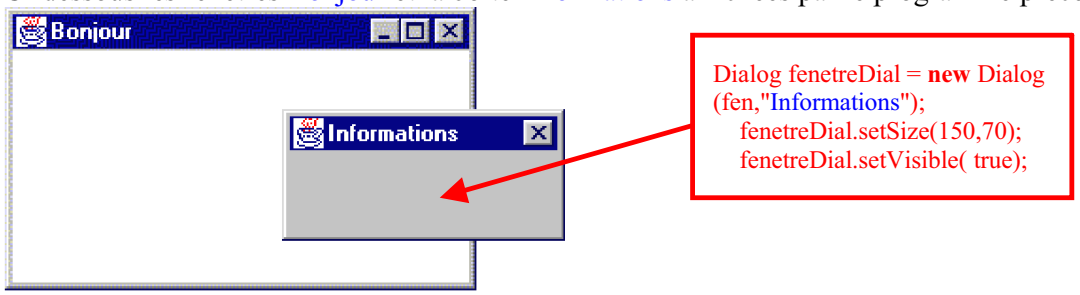
public Dialog(**Frame** owner, **String** title)
où owner est la Frame qui va appeler la boîte de dialogue, title est la string contenant le titre de la boîte de dialogue. Il faudra donc appeler le constructeur Dialog avec une Frame instanciée dans le programme.

Exemple d'affichage d'une boîte informations à partir de notre fenêtre "Bonjour" :

Programme Java

```
import java.awt.*;
class AppliWindow
{
  public static void main(String [ ] arg) {
    Frame fen = new Frame ("Bonjour" );
    fen.setBounds(100,100,250,150);
    Dialog fenetreDial = new Dialog (fen,"Informations");
    fenetreDial.setSize(150,70);
    fenetreDial.setVisible( true);
   fen. setVisible ( true );
   }
}
```

Ci-dessous les fenêtres Bonjour et la boîte Informations affichées par le programme précédent :



Composants déclenchant des actions

Ce sont essentiellement des classes directement héritées de la classe **java.awt.Container**. Les menus dérivent de la classe **java.awt.MenuComponent**. Nous ne détaillons pas tous les composants possibles, mais certains les plus utiles à créer une interface Windows-like.

Composants permettant le déclenchement d'actions :

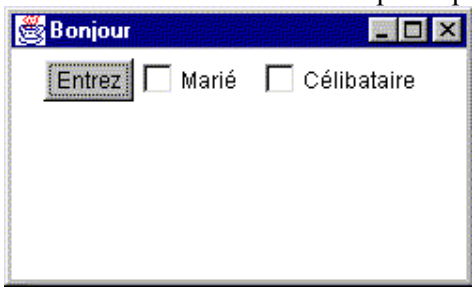
Les classes composants	Fonction
java.lang.Object | +--java.awt.MenuComponent | +--**java.awt.MenuBar**	Création d'une barre des menus dans la fenêtre.
java.lang.Object | +--java.awt.MenuComponent | +--**java.awt.MenuItem**	Création des zones de sous-menus d'un menu principal de la classique barre des menus.
java.lang.Object | +--java.awt.MenuComponent | +--java.awt.MenuItem | +--**java.awt.Menu**	Création d'un menu principal classique dans la barre des menus de la fenêtre.
java.lang.Object | +--java.awt.Component | +--**java.awt.Button**	Création d'un bouton poussoir classique (cliquable par la souris)
java.lang.Object | +--java.awt.Component | +--**java.awt.Checkbox**	Création d'un radio bouton, regroupable éventuellement avec d'autres radio boutons.

Enrichissons notre fenêtre précédente d'un bouton poussoir et de deux radio boutons :

Programme Java

```
import java.awt.*;
class AppliWindow
{
  public static void main(String [ ] arg) {
    Frame fen = new Frame ("Bonjour" );
    fen.setBounds(100,100,250,150);
    fen.setLayout(new FlowLayout( ));
    Button entree = new Button("Entrez");
    Checkbox bout1 = new Checkbox("Marié");
    Checkbox bout2 = new Checkbox("Célibataire");
    fen.add(entree);
    fen.add(bout1);
    fen.add(bout2);
    fen. setVisible ( true );
   }
}
```

Ci-dessous la fenêtre affichée par le programme précédent :



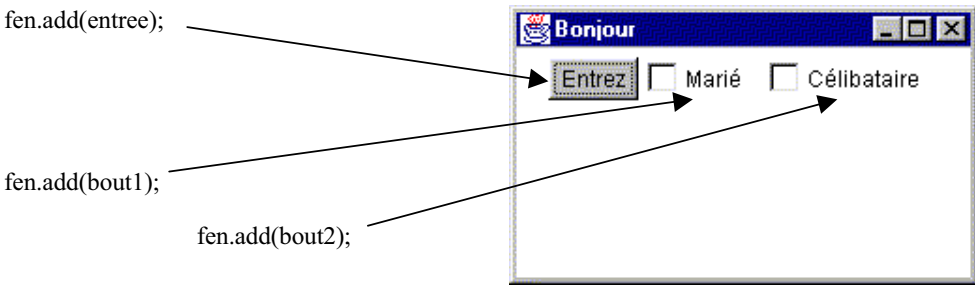
Remarques sur le programme précédent :

1) Les instructions

- `Button entree = new Button("Entrez");` → Entrez
- `Checkbox bout1 = new Checkbox("Marié");` → Marié
- `jCheckbox bout2 = new Checkbox("Célibataire");` → Célibataire

servent à **créer** un bouton poussoir (classe Button) et deux boutons radio (classe CheckBox), chacun avec un libellé.

2) Les instructions



servent à **ajouter** les objets créés au conteneur (la fenêtre fen de classe Frame).

3) L'instruction

- fen.setLayout(**new** FlowLayout( ));

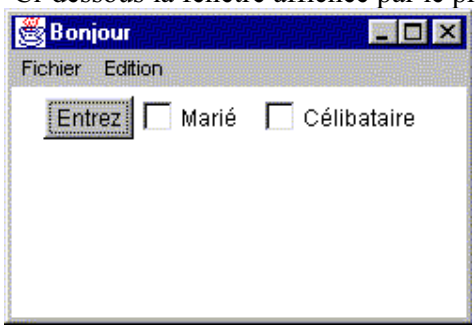
sert à **positionner** les objets visuellement dans la fenêtre les uns à côté des autres, nous en dirons un peu plus sur l'agencement visuel des composants dans une fenêtre.

Terminons la personnalisation de notre fenêtre avec l'introduction d'une barre des menus contenant deux menus : "fichier" et "édition" :

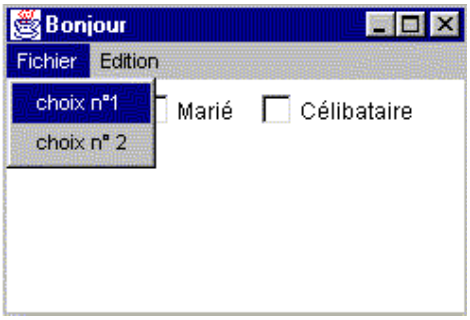
Programme Java

```
import java.awt.*;
class AppliWindow
{
  public static void main(String [ ] arg) {
    Frame fen = newFrame ("Bonjour" );
    fen.setBounds(100,100,250,150);
    fen.setLayout(new FlowLayout( ));
    Button entree = new Button("Entrez");
    Checkbox bout1 = new Checkbox("Marié");
    Checkbox bout2 = new Checkbox("Célibataire");
    fen.add(entree);
    fen.add(bout1);
    fen.add(bout2);
   // les menus :
    MenuBar mbar = new MenuBar( );
    Menu meprinc1 = new Menu("Fichier");
    Menu meprinc2 = new Menu("Edition");
    MenuItem item1 = new MenuItem("choix n°1");
    MenuItem item2 = new MenuItem("choix n° 2");
    fen.setMenuBar(mbar);
    meprinc1.add(item1);
    meprinc1.add(item2);
    mbar.add(meprinc1);
    mbar.add(meprinc2);
    fen. setVisible ( true );
   }
}
```

Ci-dessous la fenêtre affichée par le programme précédent :

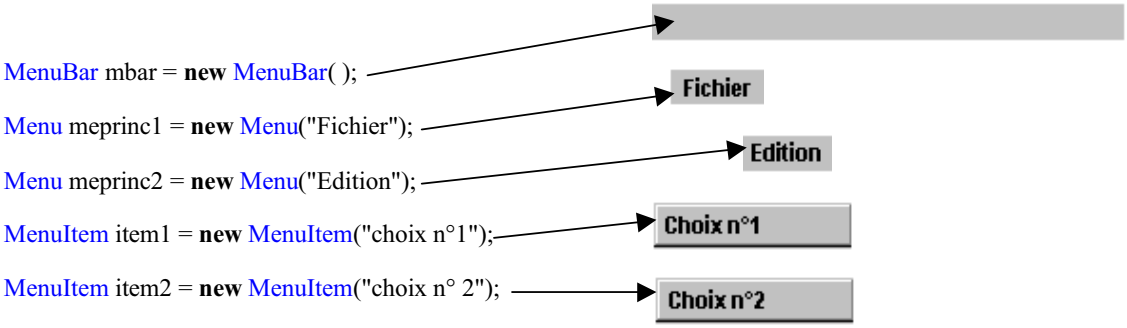


La fenêtre après que l'utilisateur clique sur le menu Fichier



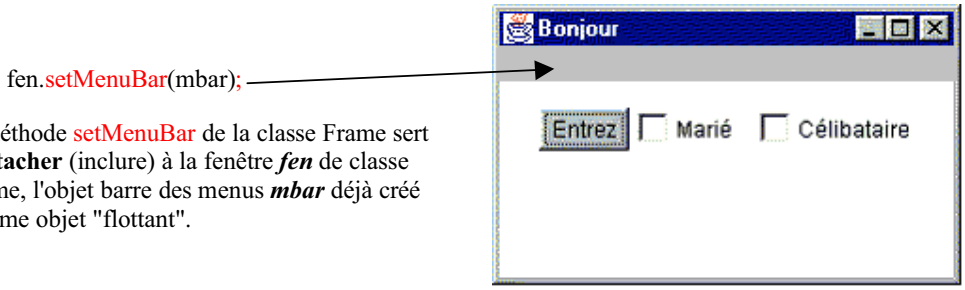
Remarques sur le programme précédent :

1) Les instructions



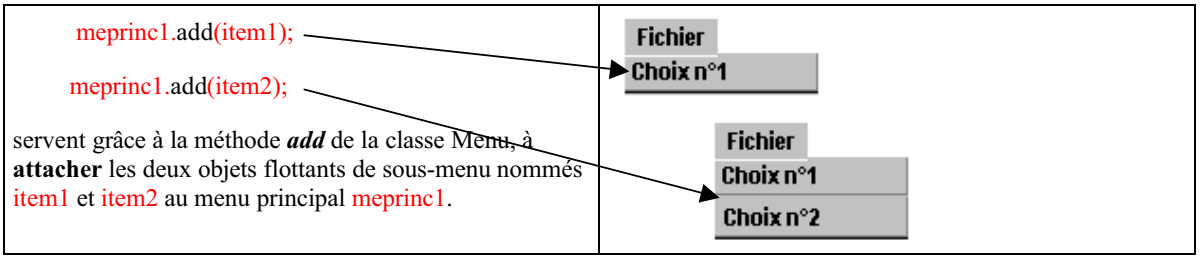
servent à **créer** une barre de menus nommée ***mbar***, deux menus principaux ***meprinc1*** et ***meprinc2***, et enfin deux sous-menus ***item1*** et ***item2*** A cet instant du programme tous ces objets existent mais ne sont pas attachés entre eux, ont peut les considérer comme des objets "flottants" .

2) Dans l'instruction



la méthode setMenuBar de la classe Frame sert à **attacher** (inclure) à la fenêtre ***fen*** de classe Frame, l'objet barre des menus ***mbar*** déjà créé comme objet "flottant".

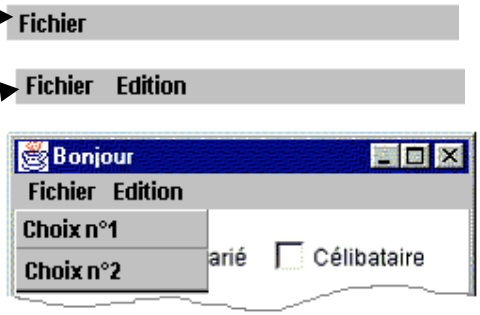
3) Les instructions



4) Les instructions

mbar.add(meprinc1);

mbar.add(meprinc2);



servent grâce à la méthode ***add*** de la classe MenuBar, à **attacher** les deux objets flottants de catégorie menu principal nommés meprinc1 et meprinc2, à la barre des menus mbar.

Remarquons enfin ici une application pratique du **polymorphisme dynamique (redéfinition)** de la méthode **add**, elle même **surchargée** plusieurs fois dans une même classe.

Composants d'affichage ou de saisie

Composants permettant l'affichage ou la saisie :

Les classes composants	Fonction
java.awt.Component | **+--java.awt.Label**	Création d'une étiquette permettant l'affichage d'un texte.
java.awt.Component | +--**java.awt.Canvas**	Création d'une zone rectangulaire vide dans laquelle l'application peut dessiner.
java.awt.Component | +--**java.awt.List**	Création d'une liste de chaînes dont chaque élément est sélectionnable.
java.awt.Component | +--java.awt.TextComponent | +--**java.awt.TextField**	Création d'un éditeur mono ligne.
java.awt.Component | +--java.awt.TextComponent | +--**java.awt.TextArea**	Création d'un éditeur multi ligne.

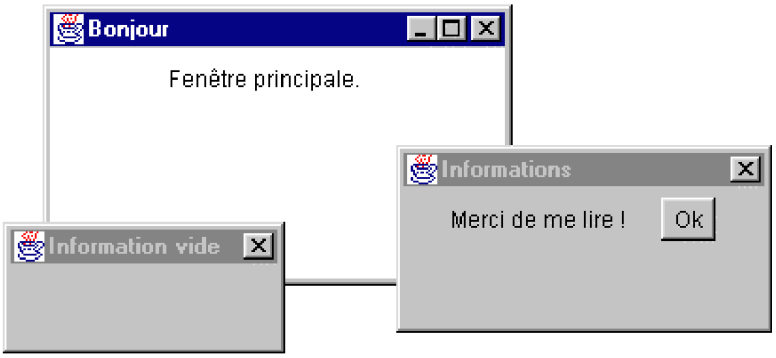
Ces composants s'ajoutent à une fenêtre après leurs créations, afin d'être visible sur l'écran comme les composants de Button, de CheckBox, etc...

Ces composants sont à rapprocher quant à leurs fonctionnalités aux classes Delphi de composant standards, nous en donnons la correspondance dans le tableau ci-dessous :

Les classes Java	Les classes Delphi
java.awt.Label	**TLabel**
java.awt.Canvas	**TCanvas**
java.awt.List	**TListBox**
java.awt.TextField	**TEdit**
java.awt.TextArea	**TMemo**
java.awt.CheckBox	**TCheckBox**
java.awt.Button	**TButton**

Exemple récapitulatif :

Soit à afficher une fenêtre principale contenant le texte "fenêtre principal" et deux fenêtres de dialogue, l'une vide directement instancié à partir de la classe Dialog, l'autre contenant un texte et un bouton, instanciée à partir d'une classe de boîte de dialogue personnalisée. L'exécution du programme produira le résultat suivant :



Nous allons construire un programme contenant **deux classes**, la première servant à définir le genre de boîte personnalisée que nous voulons, la seconde servira à créer une boîte vide et une boîte personnalisée et donc à lancer l'application.

Première classe :

La classe de dialoque personnalisée

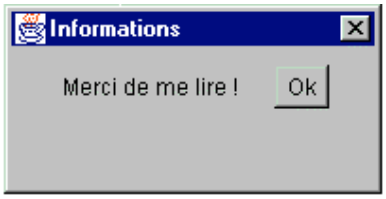
```
import java.awt.*;
class  UnDialog extends Dialog
{
  public UnDialog(Frame mere)
  {
   super(mere,"Informations");
```

```
    Label etiq = new Label("Merci de me lire !");
    Button bout1 = new Button("Ok");
    setSize(200,100);
    setLayout(new FlowLayout( ));
    add(etiq);
    add(bout1);
    setVisible ( true );
   }
}
```

Explications pas à pas des instructions :

Cette classe ***UnDialog*** ne contient que le constructeur permettant d'instancier des objets de cette classe, elle dérive (hérite) de la classe Dialog < **class** UnDialog **extends** Dialog >

On appelle immédiatement le constructeur de la classe mère (Dialog) par l'instruction < **super**(mere,"Informations"); > on lui fournit comme paramètres : la Frame propriétaire de la boîte de dialogue et le titre de la future boîte. On crée une Label <Label etiq = **new** Label("Merci de me lire !");>, On crée un Button <Button bout1 = **new** Button("Ok");>. On définit la taille de la boîte à instancier <setSize(200,100);> On indique le mode d'affichage des composants qui y seront déposés <setLayout(new FlowLayout( ));>	

On ajoute la Label <add(etiq);> à la future boîte, On ajoute le Button <add(bout1);> La future boîte devra s'afficher à la fin de sa création <setVisible (true);>	

Seconde classe :
Une classe principale servant à lancer l'application et contenant la méthode main :

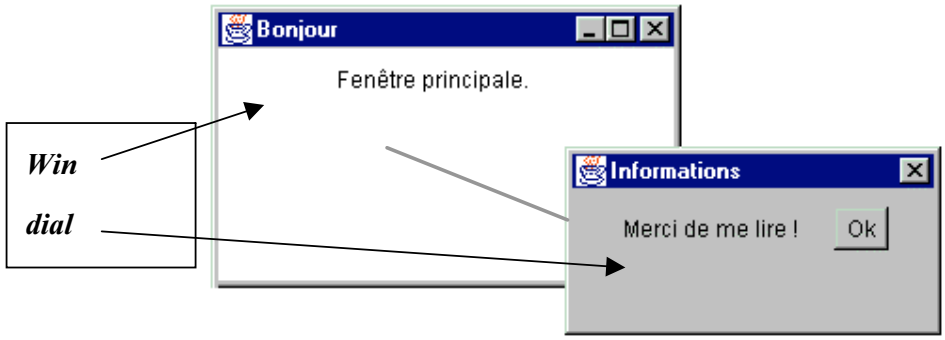
La classe principale contenant main

```
class  AppliDialogue
{
  public static void main(String [] arg) {
   Frame win = new Frame("Bonjour");
   UnDialog dial = new UnDialog(win);
   Dialog dlg = new Dialog(win,"Information vide");
   dlg.setSize(150,70);
   dlg. setVisible ( true );
   win.setBounds(100,100,250,150);
   win.setLayout(new FlowLayout( ));
   win.add(new Label("Fenêtre principale."));
   win. setVisible ( true );
  }
```

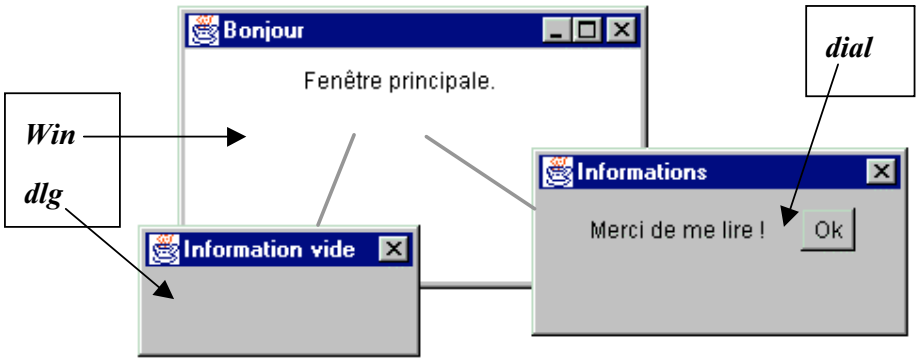
```
}
```

Toutes les instructions de la méthode main mise à part l'instruction <UnDialog dial = **new** UnDialog(win);>, correspondent à ce que nous avons écrit plus haut en vue de la création d'une fenêtre ***win*** de classe Frame dans laquelle nous ajoutons une Label et qui lance une boîte de dialogue ***dlg*** :

L'instruction <UnDialog dial = **new** UnDialog(win);> sert à instancier un objet ***dial*** de notre classe personnalisée, cet objet étant rattaché à la fenêtre ***win*** :



L'instruction <Dialog dlg = **new** Dialog(win,"Information vide");> sert à instancier un objet ***dlg*** de classe générale Dialog, cet objet est aussi rattaché à la fenêtre ***win*** :



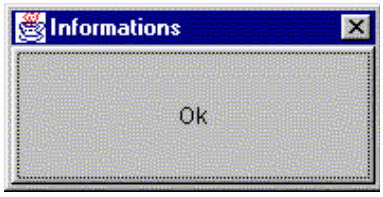
Le programme Java avec les 2 classes

```
import java.awt.*;
class  UnDialog extends Dialog {
  public UnDialog(Frame mere)
  {
   super(mere,"Informations");
        ....../// instructions
   setVisible ( true );
  }
}
class  AppliDialogue {
  public static void main(String [] arg) {
   Frame win = new Frame("Bonjour");
      ....../// instructions
   win.setVisible ( true );
  }
```

```
}
```

Comment gérer la position d'un composant dans un conteneur de classe Container : Le Layout Manager

En reprenant la fenêtre de dialogue précédente, observons l'effet visuel produit par la présence ou non d'un **Layout Manager** :

La classe de dialogue sans Layout Manager	
import java.awt.*; **class** AppliUnDialog2 **extends** Dialog { **public** AppliUnDialog2(Frame mere) { super(mere,"Informations"); Label etiq = **new** Label("Merci de me lire !"); Button bout1 = **new** Button("Ok"); setSize(200,100); *//setLayout(new FlowLayout( ));* add(etiq); add(bout1); setVisible (true); } **public static void** main(String[] args) { Frame fen = **new** Frame("Bonjour"); AppliUnDialog2 dlg = **new** AppliUnDialog2(fen); } }	Voici ce que donne l'exécution de ce programme Java. En fait lorsqu'aucun Layout manager n'est spécifié, c'est par défaut la classe du Layout <BorderLayout> qui est utilisée par Java. Cette classe n'affiche qu'un seul élément en une position fixée.  Nous remarquons que le bouton masque l'étiquette en prenant toute la place.

Soit les instructions d'ajout des composants dans le constructeur **public** AppliUnDialog2(Frame mere)	**Intervertissons** l'ordre d'ajout du bouton et de l'étiquette, toujours en laissant Java utiliser le <BorderLayout> par défaut :
add(etiq); add(bout1); setVisible (true);	add(bout1); add(etiq); setVisible (true);

voici l'effet visuel obtenu :

Cette fois c'est l'étiquette (ajoutée en dernier) qui masque le bouton !

Définissons un autre Layout puisque celui-ci ne nous plaît pas, utilisons la classe <FlowLayout> qui place les composants les uns à la suite des autres de la gauche vers la droite, l'affichage visuel continuant à la ligne suivante dès que la place est insuffisante. L'instruction <setLayout(**new** FlowLayout( ));>, assure l'utilisation du FlowLayout pour notre fenêtre de dialogue.

La classe de dialogue avec FlowLayout

```
import java.awt.*;
class  AppliUnDialog2 extends Dialog
{
   public AppliUnDialog2(Frame mere)
   {
       super(mere,"Informations");
       Label etiq = new Label("Merci de me lire !");
       Button bout1 = new Button("Ok");
       setSize(200,100);
       setLayout(new FlowLayout( ));
       add(etiq);
       add(bout1);
      setVisible ( true );
   }
  public static void main(String[ ] args) {
     Frame fen = new Frame("Bonjour");
     AppliUnDialog2 dlg = new AppliUnDialog2(fen);
  }
}
```

voici l'effet visuel obtenu :

Si comme précédemment l'on échange l'ordre des instructions d'ajout du bouton et de l'étiquette :

```
setLayout(new FlowLayout( ));
add(bout1);
add(etiq);
```

on obtient l'affichage inversé :

D'une manière générale, utilisez la méthode < **public void** setLayout(LayoutManager mgr) > pour indiquer quel genre de positionnement automatique (cf. aide du JDK pour toutes possibilités) vous conférez au Container (ici la fenêtre) votre façon de gérer le positionnement des composants de la fenêtre. Voici à titre d'information tirées du JDK, les différentes façons de positionner un composant dans un container.

héritant de LayoutManager :
GridLayout, FlowLayout, ViewportLayout, ScrollPaneLayout, BasicOptionPaneUI.ButtonAreaLayout, BasicTabbedPaneUI.TabbedPaneLayout, BasicSplitPaneDivider.DividerLayout, BasicInternalFrameTitlePane.TitlePaneLayout, BasicScrollBarUI, BasicComboBoxUI.ComboBoxLayoutManager, BasicInternalFrameUI.InternalFrameLayout.

héritant de LayoutManager2 :
CardLayout, GridBagLayout, BorderLayout, BoxLayout, JRootPane.RootLayout, OverlayLayout, BasicSplitPaneUI.BasicHorizontalLayoutManager.

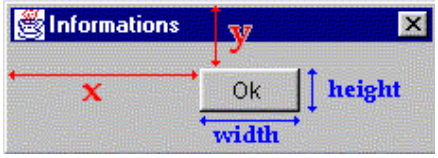
Vous notez qu'il est impossible d'être exhaustif sans devenir assommant, à chacun d'utiliser les Layout en observant leurs effets visuels.

Il est enfin possible, si aucun des Layout ne vous convient de gérer personnellement au pixel près la position d'un composant. Il faut tout d'abord indiquer que vous ne voulez aucun Layoutmanager, puis ensuite préciser les coordonnées et la taille de votre composant.

Indiquer qu'aucun Layout n'est utilisé : setLayout(**null**); //on passe la référence null comme paramètre à la méthode de définition du Layout

Préciser les coordonnées et la taille du composant avec sa méthode setBounds :
public void setBounds(**int** x, **int** y, **int** width, **int** height)

Exemple, les paramètres de setBounds pour un Button :

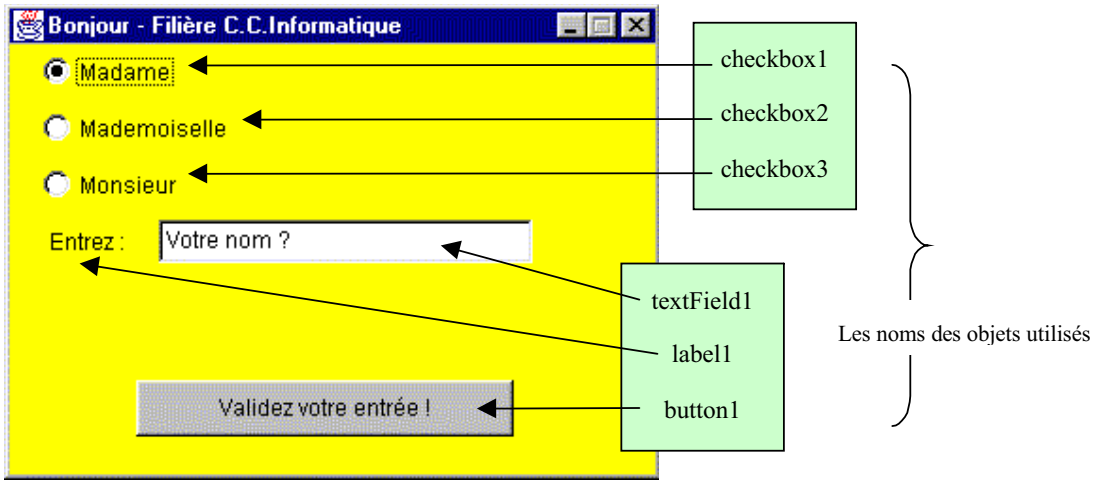


Si nous voulons positionner nous mêmes un composant ***Component*** *comp* dans la fenêtre, nous utiliserons la méthode add indiquant le genre de façon de ranger ce composant (LayoutManager)

public void add(Component comp,Object constraints)
add(checkbox1, **new** FlowLayout( ));
ou bien
add(checkbox1, **null**);

Une application fenêtrée pas à pas

Nous construisons une IHM de saisie de renseignements concernant un(e) étudiant(e) :



ci-après le code Java du programme :

```
class AppliIHM { //  classe principale
  //Méthode principale
  public static void main(String[] args) { // lance le programme
     Cadre1 fenetre = new Cadre1( );// création d'un objet de classe Cadre1
     fenetre.setVisible(true);// cet objet de classe Cadre1 est rendu visible sur l'écran
  }
}

import java.awt.*; // utilisation des classes du package awt
class Cadre1 extends Frame { // la classe Cadre1 hérite de la classe des fenêtres Frame
  Button button1 = new Button( );// création d'un objet de classe Button
  Label label1 = new Label( );// création d'un objet de classe Label
  CheckboxGroup checkboxGroup1 = new CheckboxGroup( );// création d'un objet groupe de checkbox
  Checkbox checkbox1 = new Checkbox( );// création d'un objet de classe Checkbox
  Checkbox checkbox2 = new Checkbox( );// création d'un objet de classe Checkbox
  Checkbox checkbox3 = new Checkbox( );// création d'un objet de classe Checkbox
  TextField textField1 = new TextField( );// création d'un objet de classe TextField

  //Constructeur de la fenêtre
  public Cadre1( ) {  //Constructeur sans paramètre
   Initialiser( );// Appel à une méthode privée de la classe
  }

  //Initialiser la fenêtre :
  private void Initialiser( ) {  //Création et positionnement de tous les composants
    this.setResizable(false); // la fenêtre ne peut pas être retaillée par l'utilisateur
    this.setLayout(null); // pas de Layout, nous positionnons les composants nous-mêmes
    this.setBackground(Color.yellow); // couleur du fond de la fenêtre
    this.setSize(348, 253); // widht et height de la fenêtre
    this.setTitle("Bonjour - Filière C.C.Informatique"); // titre de la fenêtre
    this.setForeground(Color.black); // couleur de premier plan de la fenêtre
    button1.setBounds(70, 200, 200, 30); // positionnement du bouton
    button1.setLabel("Validez votre entrée !"); // titre du bouton
    label1.setBounds(24, 115, 50, 23); // positionnement de l'étiquette
    label1.setText("Entrez :"); // titre de l'étiquette
    checkbox1.setBounds(20, 25, 88, 23); // positionnement du CheckBox
    checkbox1.setCheckboxGroup(checkboxGroup1); // ce CheckBox est mis dans le groupe checkboxGroup1
    checkbox1.setLabel("Madame");// titre du CheckBox
    checkbox2.setBounds(20, 55, 108, 23);// positionnement du CheckBox
    checkbox2.setCheckboxGroup(checkboxGroup1);// ce CheckBox est mis dans le groupe checkboxGroup1
    checkbox2.setLabel("Mademoiselle");// titre du CheckBox
    checkbox3.setBounds(20, 85, 88, 23);// positionnement du CheckBox
    checkbox3.setCheckboxGroup(checkboxGroup1);// ce CheckBox est mis dans le groupe checkboxGroup1
    checkbox3.setLabel("Monsieur");// titre du CheckBox
    checkboxGroup1.setSelectedCheckbox(checkbox1);// le CheckBox1 du groupe est coché au départ
    textField1.setBackground(Color.white);// couleur du fond de l'éditeur mono ligne
    textField1.setBounds(82, 115, 198, 23);// positionnement de l'éditeur mono ligne
    textField1.setText("Votre nom ?");// texte de départ de l'éditeur mono ligne
    this.add(checkbox1);// ajout dans la fenêtre du CheckBox
    this.add(checkbox2);// ajout dans la fenêtre du CheckBox
    this.add(checkbox3);// ajout dans la fenêtre du CheckBox
    this.add(button1);// ajout dans la fenêtre du bouton
    this.add(textField1);// ajout dans la fenêtre de l'éditeur mono ligne
    this.add(label1);// ajout dans la fenêtre de l'étiquette
  }
}
```

Maintenant que nous avons construit la partie affichage de l'IHM, il serait bon qu'elle interagisse

avec l'utilisateur, par exemple à travers des messages comme les événements de souris ou bien d'appui de touches de claviers. Nous allons voir comment Java règle la gestion des échanges de messages entre le système et votre application.

Les événements

Rappelons ce que nous connaissons de la programmation par événements (cf.package chap.5.2)

Principes de la programmation par événements

La programmation événementielle :

Logique selon laquelle un programme est construit avec des objets et leurs propriétés et d'après laquelle seules les interventions de l'utilisateur sur les objets du programme déclenchent l'exécution des routines associées.

Avec des systèmes multi-tâches préemptifs sur micro-ordinateur , le système d'exploitation passe l'essentiel de son " temps " à **attendre une action de l'utilisateur** (événement). Cette action **déclenche un message** que le système traite et envoie éventuellement à une application donnée.

Nous pourrons construire un logiciel qui réagira sur les interventions de l'utilisateur si nous arrivons à récupérer dans notre application les messages que le système envoie. Nous avons déjà utilisé l'environnement Delphi de Borland, et Visual Basic de Microsoft, Java autorise aussi la consultation de tels messages.

- *L'**approche événementielle*** intervient principalement dans l'interface entre le logiciel et l'utilisateur, mais aussi dans la liaison dynamique du logiciel avec le système, et enfin dans la sécurité.
- *L'**approche visuelle*** nous aide et simplifie notre tâche dans la construction du dialogue homme-machine.

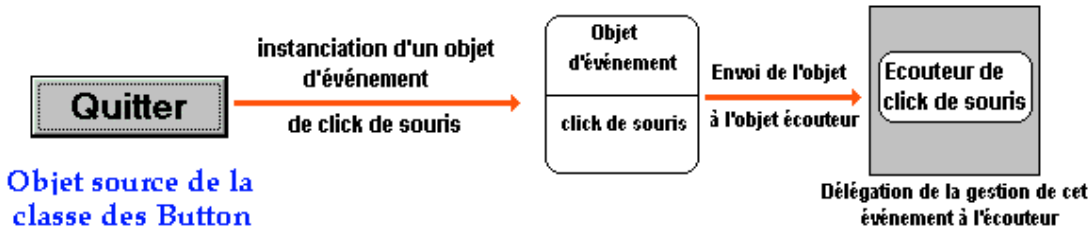
La combinaison de ces deux approches produit un logiciel habillé et adapté au système d'exploitation.

Il est possible de relier certains objets entre eux par des relations événementielles. Nous les représenterons par un graphe (structure classique utilisée pour représenter des relations).

Modèle de délégation de l'événement en Java

En Java, le traitement et le transport des messages associés aux événements sont assurés par deux objets dans le cadre d'un modèle de communication dénommé le modèle de traitement des événements par délégation (Delegation Event Model) :

Le message est envoyé par une source ou **déclencheur** de l'événement qui sera un composant Java, à un récepteur ou **écouteur** de l'événement qui est **chargé de gérer l'événement**, ce sera un objet de la classe des écouteurs instancié et ajouté au composant :



La méthode de programmation de l'interception des événements est nettement plus lourde syntaxiquement en Java qu'en Delphi et en Visual Basic, mais elle est permet beaucoup plus de choix et elle est entièrement objet. Ce sont des classes abstraites dont le nom généralement se termine par **Listener**. Chacune de ces classes étend la classe abstraite d'interface **EventListener**. Toutes ces classes d'écouteurs d'événements sont situées dans le package **java.awt.event**, elles se chargent de fournir les méthodes adéquates aux traitements d'événements envoyés par un déclencheur.

Voici la liste des interfaces d'écouteurs d'événements extraite du JDK 1.4.2
Action, ActionListener, AdjustmentListener, AncestorListener, AWTEventListener, BeanContextMembershipListener, BeanContextServiceRevokedListener, BeanContextServices, BeanContextServicesListener, CaretListener, CellEditorListener, ChangeListener, ComponentListener, ContainerListener, DocumentListener, DragGestureListener, DragSourceListener, DropTargetListener, FocusListener, HyperlinkListener, InputMethodListener, InternalFrameListener, ItemListener, KeyListener, ListDataListener, ListSelectionListener, MenuDragMouseListener, MenuKeyListener, MenuListener, MouseInputListener, MouseListener, MouseMotionListener, PopupMenuListener, PropertyChangeListener, TableColumnModelListener, TableModelListener, TextListener, TreeExpansionListener, TreeModelListener, TreeSelectionListener, TreeWillExpandListener, UndoableEditListener, VetoableChangeListener, WindowListener.

Les événements possibles dans Java sont des objets (un événement est un message contenant plusieurs informations sur les états des touches de clavier, des paramètres,...) dont les classes sont dans le package **java.awt.event.**

Voici quelques classes générales d'événements possibles tirées du JDK 1.4.2:
ActionEvent, AdjustmentEvent, AncestorEvent, ComponentEvent, InputMethodEvent, InternalFrameEvent, InvocationEvent, ItemEvent, TextEvent.

Intercepter un click de souris sur un bouton

Supposons avoir défini le bouton : Button bouton = **new** Button("Entrez");

Il nous faut choisir une classe d'écouteur afin de traiter l'événement click de souris. Pour intercepter un click de souris nous disposons de plusieurs moyens, c'est ce qui risque de dérouter le débutant. Nous pouvons en fait l'intercepter à deux niveaux.

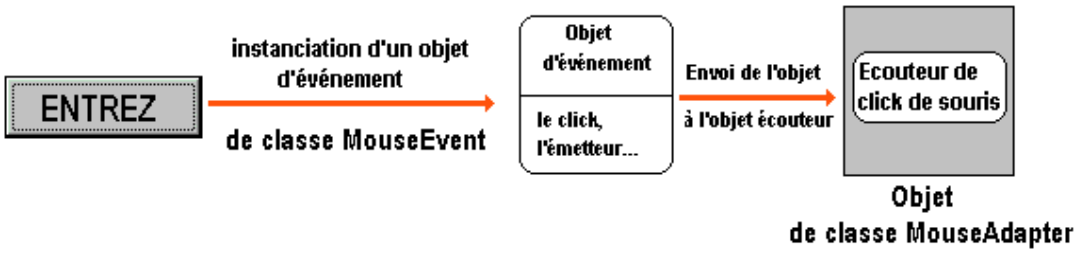
Interception de bas niveau :

Les classes précédentes se dérivent en de nombreuses autres sous-classes. Par exemple, la classe **MouseEvent** qui encapsule tous les événements de souris de **bas niveau**, dérive de la classe ComponentEvent :

```
+--java.awt.event.ComponentEvent
                |
                +--java.awt.event.InputEvent
                     |
                     +--java.awt.event.MouseEvent
```

Nous pourrons par exemple, choisir l'interface MouseListener (abstraite donc non instanciable, mais implémentable) dont la fonction est d'intercepter (écouter) les événements de souris (press, release, click, enter, et exit).

Il existe une classe abstraite implémentant l'interface MouseListener qui permet d'instancier des écouteurs de souris, c'est la classe des MouseAdapter.

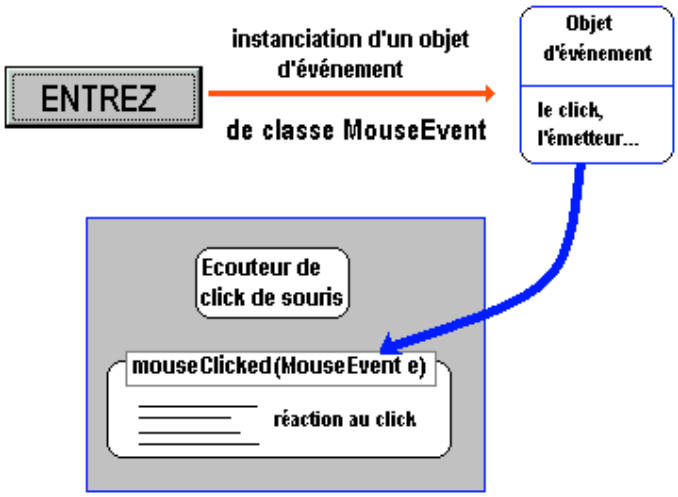


Dans ce cas il suffit de redéfinir la méthode de la classe MouseAdapter qui est chargée d'intercepter et de traiter l'événement qui nous intéresse (cet événement lui est passé en paramètre):

Méthode à redéfinir	Action déclenchant l'événement
void mouseClicked(MouseEvent e)	invoquée lorsqu'il y a eu un click de souris sur le composant.
void mouseEntered(MouseEvent e)	invoquée lorsque la souris entre dans le rectangle visuel du composant.

void mouseExited(MouseEvent e)	invoquée lorsque la souris sort du rectangle visuel du composant.
void mousePressed(MouseEvent e)	invoquée lorsqu'un des boutons de la souris a été appuyé sur le composant.
void mouseReleased(MouseEvent e)	invoquée lorsqu'un des boutons de la souris a été relâché sur le composant.

L'événement est passé en paramètre de la méthode : mouseClicked(MouseEvent e)



Démarche pratique pour gérer le click du bouton

Construire une classe InterceptClick héritant de la classe abstraite MouseAdapter et redéfinir la méthode mouseClicked :	**class** InterceptClick **extends** MouseAdapter { **public void** mouseClicked(MouseEvent e) { *//.... actions à exécuter.* } }
Ensuite nous devons instancier un objet écouteur de cette classe InterceptClick :	InterceptClick clickdeSouris = **new** InterceptClick( );
Enfin nous devons ajouter cet écouteur à l'objet bouton :	bouton.addMouseListener(clickdeSouris);

Les étapes 2° et 3° peuvent être recombinées en une seule étape:

```
bouton.addMouseListener( new InterceptClick( ) );
```

Remarque :
Afin de simplifier encore plus l'écriture du code, Java permet d'utiliser ici une **classe anonyme** (classe locale sans nom) comme paramètre effectif de la méthode addMouseListener. On ne déclare pas de nouvelle classe implémentant la classe abstraite MouseAdapter, mais on la définie **anonymement** à l'intérieur de l'appel au constructeur.

Les étapes 1°, 2° et 3° peuvent être alors recombinées en une seule, nous comparons ci-dessous l'écriture avec une classe anonyme :

Classe anonyme	Classe dérivée de MouseAdapter
Méthode xxx : bouton.addMouseListener (**new** MouseAdapter( ) { **public void** mouseClicked(MouseEvent e) {//.... *actions à exécuter.* } }); la référence à l'objet d'écouteur n'est pas accessible.	**class** InterceptClick **extends** MouseAdapter { **public void** mouseClicked(MouseEvent e) { //.... *actions à exécuter.* } } Méthode xxx : InterceptClick clickdeSouris = **new** InterceptClick( ); bouton.addMouseListener(clickdeSouris);
La classe anonyme est recommandée lorsque la référence à l'objet d'écouteur n'est pas utile. On se trouve dans le cas semblabe à Delphi où l'écouteur est l'objet de bouton lui-même.	

Interception de haut niveau ou sémantique :

Sun a divisé d'une façon très artificielle les événements en deux catégories : les événements de bas niveau et les événements sémantiques : Les événement de bas niveau représentent des événements système de gestion de fenêtre de périphérique, souris, clavier et les entrées de bas niveau, tout le reste est événement sémantique.

Toutefois, Java considère qu'un click de souris sur un bouton qui est une action particulière de bas niveau, est aussi une action sémantique du bouton.

Il existe une classe d'événement générique qui décrit tous les autres événements dont le cas particulier du **click de souris sur un bouton**, c'est la classe **java.awt.event.ActionEvent**. Un événement est donc un objet instancié de la classe **ActionEvent**, cet événement générique est passé à des écouteurs génériques de l'interface ActionListener, à travers l'ajout de l'écouteur au composant par la méthode addActionListener.

Nous allons donc reprendre la programmation de notre objet bouton de la classe des Button avec cette fois-ci un écouteur de plus haut niveau : un objet construit à partir d'implementation de l'interface ActionListener.

L'interface ActionListener, n'a aucun attribut et ne possède qu'une seule méthode à redéfinir et traitant l'événement **ActionEvent** :

la Méthode à redéfinir	Action déclenchant l'événement
public void actionPerformed (ActionEvent e)	Toute action possible sur le composant.

Nous pouvons comme dans le traitement par un événement de bas niveau, décomposer les lignes de code en créant une classe implémentant la classe abstraite des ActionListener, ou bien créer une classe anonyme. La démarche étant identique à l'interception de bas niveau, nous livrons directement ci-dessous les deux programmes Java équivalents :

Version avec une classe implémentant ActionListener

```
import java.awt.*;
import java.awt.event.*;

class EventHigh implements ActionListener
{ public void actionPerformed(ActionEvent e)
  {  //.... actions à exécuter.
  }
}
class ApplicationEventHigh
{
   public static void main(String [ ] arg)
   { ....
    Button bouton = new Button("Entrez");
    bouton.addActionListener(newEventHigh( ));
    .... }
}
```

Version avec une classe anonyme

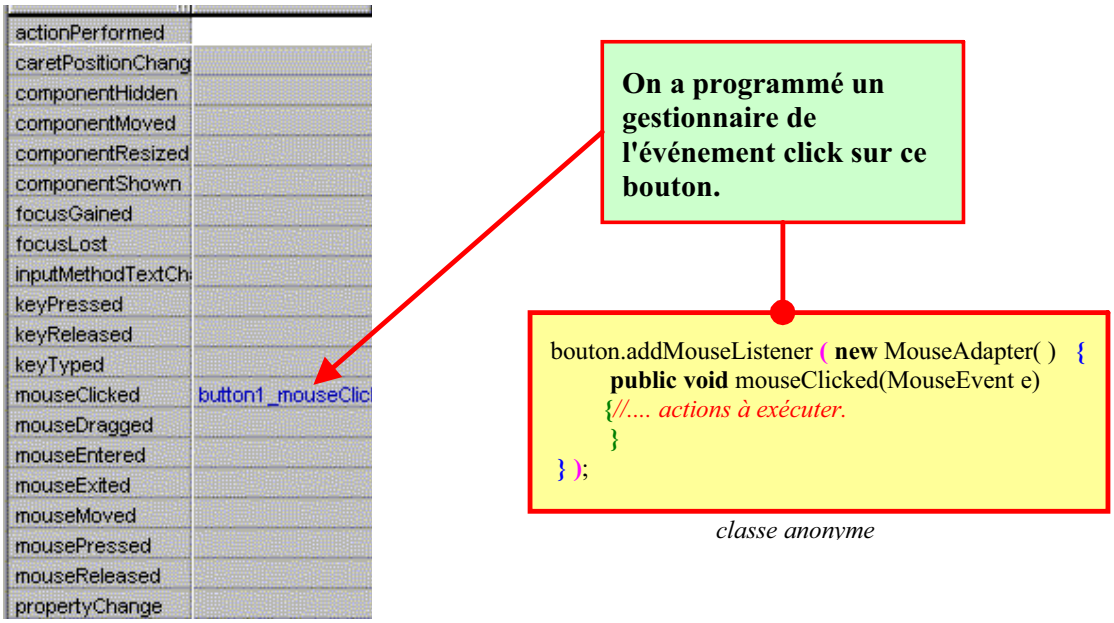
```
  import java.awt.*;
  import java.awt.event.*;

class ApplicationEventHigh
{
  public static void main(String [ ] arg)
  { ....
   Button bouton = new Button("Entrez");
   bouton.addActionListener(new EventHigh( )
    {public void actionPerformed (ActionEvent e)
     {//.... actions à exécuter.
     }
    } );
   ....
  }
}
```

Nous voyons sur ce simple exemple, qu'il est impossible d'être exhaustif tellement les cas particuliers foisonnent en Java, aussi allons nous programmer quelques interceptions d'événements correspondant à des situations classiques. Les évolutions sont nombreuses depuis la version 1.0 du JDK et donc seuls les principes sont essentiellement à retenir dans notre approche.

En outre, tous les objets de composants ne sont pas réactifs à l'ensemble de tous les événements existants, ce qui nécessite la connaissance des relations possibles pour chaque composant. Cet apprentissage est facilité par des outils qui classifient les événements par objet et engendrent le squelette du code du traitement à effectuer pour chaque événement.

La construction d'une IHM efficace en Java, s'effectuera avec un **RAD comme JBuilder équivalent Delphi pour Java ou NetBeans de Sun**, qui génère automatiquement les lignes de codes nécessaires à l'interception d'événements et donc simplifie l'apprentissage et la tâche du développeur !

Voici regroupés dans JBuilder la liste des événements auquel un bouton (objet de classe Button) est sensible :



Vous remarquerez que **actionPerformed** et **mouseClicked** sont les méthodes avec lesquelles nous traiterons l'événement click soit en haut niveau, soit en bas niveau. JBuilder agissant comme générateur de code, construira automatiquement les classes anonymes associées à votre choix.

Appliquons la démarche que nous venons de proposer à un exemple exécutable.

Terminer une application par un click de bouton

Pour arrêter la machine virtuelle Java et donc terminer l'application qui s'exécute, il faut utiliser la méthode **exit( )** de la classe System. Nous programmons cette ligne d'arrêt lorsque l'utilisateur clique sur un bouton présent dans la fenêtre à l'aide de l'événement de haut niveau..

1°) Implémenter une classe héritant de la classe abstraite des ActionListener :
Cette classe ActionListener ne contient qu'une seule méthode < **public void** actionPerformed(ActionEvent e) > dont la seule fonction est d'être invoquée dès qu'un événement quelconque est transmis à l'objet ActionListener à qui elle appartient (objet à ajouter au composant), cette fonction est semblable à celle d'un super gestionnaire générique d'événement et c'est dans le corps de cette méthode que vous écrivez votre code. Comme la classe ActionListener est abstraite, on emploi le mot clef **implements** au lieu de **extends** pour une classe dérivée.
Nous devons redéfinir (**surcharge dynamique**) la méthode actionPerformed(ActionEvent e) avec notre propre code :

Classe dérivée de ActionListener

```
import java.awt.*;
import java.awt.event.*;

 class ListenerQuitter implements ActionListener
 { public void actionPerformed(ActionEvent e)
   { System.exit(0); // arrêter la machine java
   }
 }
```

2°) Instancier et ajouter un objet de la classe héritant de ActionListener :

Un objet de cette classe ListenerQuitter doit être créé pour être ensuite ajouté dans le composant qui sera chargé de fermer l'application :

```
ListenerQuitter gestionbouton = new ListenerQuitter( );
```

Cet objet maintenant créé peut être ajouté au composant qui lui enverra l'événement. Cet ajout a lieu grâce à la méthode addActionListener de la classe des composants : (par exemple ajouter ce gestionnaire à Button Unbouton) :

```
Button Unbouton;
Unbouton.addActionListener(gestionbouton);
```

Les deux actions précédentes pouvant être combinées en une seule équivalente:

```
Unbouton.addActionListener( new ListenerQuitter( ));
```

Méthode main

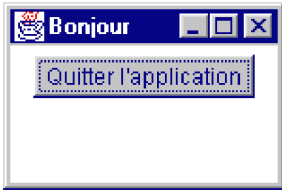
```
public static void main(String [] arg) {
    Frame fen = new Frame ("Bonjour" );
    fen.setBounds(100,100,150,80);
    fen.setLayout(new FlowLayout( ));
    Button quitter = new Button("Quitter l'application");
    quitter.addActionListener(new ListenerQuitter( ));
    fen.add(quitter);
    fen.setVisible(true);
  }
```

Le programme Java complet

```
import java.awt.*;
import java.awt.event.*;

 class ListenerQuitter implements ActionListener
 { public void actionPerformed(ActionEvent e)
   { System.exit(0); // arrêter la machine java
   }
 }
 class AppliBoutonQuitter
 {
   public static void main(String [] arg) {
```

```
        Frame fen = new Frame ("Bonjour" );
        fen.setBounds(100,100,150,80);
        fen.setLayout(new FlowLayout( ));
        Button quitter = new Button("Quitter l'application");
        quitter.addActionListener(new ListenerQuitter( ));
        fen.add(quitter);
        fen.setVisible(true);
      }
   }
```

La fenêtre associée à ce programme :

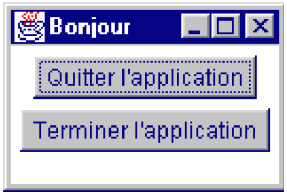


Voici une version de la méthode main du programme précédent dans laquelle nous affichons un deuxième bouton "Terminer l'application" auquel nous avons ajouté le même gestionnaire de fermeture de l'application :

Méthode main

```
public static void main(String [] arg) {
    Frame fen = new Frame ("Bonjour" );
    fen.setBounds(100,100,150,80);
    fen.setLayout(new FlowLayout( ));
    ListenerQuitter obj =  new ListenerQuitter( );
    Button quitter = new Button("Quitter l'application");
    Button terminer = new Button("Terminer l'application");
    quitter.addActionListener(obj);
    terminer.addActionListener(obj);
    fen.add(quitter);
    fen.add(terminer);
    fen.setVisible(true);
  }
```

Les deux boutons exécutent la même action : arrêter l'application



Java permet d'utiliser, comme nous l'avons indiqué plus haut, une **classe anonyme** (classe locale sans nom) comme paramètre effectif de la méthode addActionListener.

Au lieu d'écrire :
terminer.addActionListener(new ListenerQuitter( ));

La classe anonyme remplaçant tout le code :

```
terminer.addActionListener(new ActionListener( )   {
     public void actionPerformed(ActionEvent e)
     {
       System.exit(0);
     }
   }
);
```

Nous conseillons au lecteur de reprogrammer cet exemple à titre d'exercice, avec l'événement click de bas niveau.

Intérêt d'implémenter une interface XXXListener

Un événement est donc un message constitué suite à une action qui peut survenir à tout moment et dans divers domaines (click de souris, clavier,...), cela dépendra uniquement de l'objet source qui est le déclencheur de l'événement.

Nous allons à partir d'un bouton accéder à d'autres composants présents sur la même fiche, pour cela nous passerons en paramètre au constructeur de la classe implémentant l'interface ActionListener les objets à modifier lors de la survenue de l'événement.

L'utilisation d'une telle classe **class** ListenerGeneral **implements** ActionListener est évident : nous pouvons rajouter à cette classe des champs et des méthodes permettant de personnaliser le traitement de l'événement.

Soit au départ l'interface suivante :

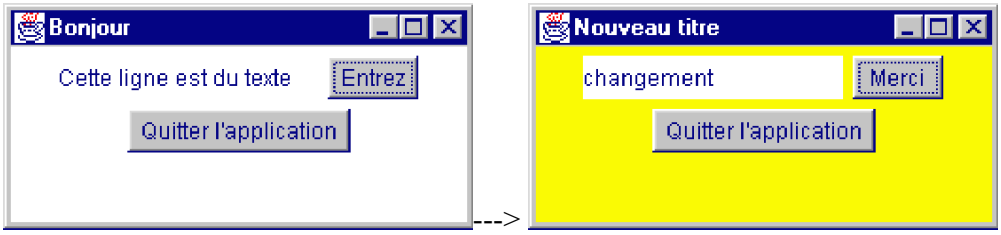
Nous programmons :

- Lorsque l'utilisateur clique sur le bouton "Quitter l'application":
    - la fermeture de la fenêtre et l'arrêt de l'application ,
- Lorsque l'utilisateur clique sur le bouton "Entrez":
    - le changement de couleur du fond de la fiche,
    - le changement du texte de l'étiquette,
    - le changement de libellé du bouton,
    - le changement du titre de la fenêtre.

Le programme Java complet

```
import java.awt.*;
import java.awt.event.*;

 class ListenerGeneral implements ActionListener
 { Label etiq;
   Frame win;
   Button bout;
  //constructeur :
   public ListenerGeneral(Button bouton, Label etiquette, Frame window)
   { this.etiq = etiquette;
     this.win = window;
     this.bout = bouton;
    }
   public void actionPerformed(ActionEvent e)
    // Actions sur l'étiquette, la fenêtre, le bouton lui-même :
   { etiq.setText("changement");
     win.setTitle ("Nouveau titre");
     win.setBackground(Color.yellow);
     bout.setLabel("Merci");
   }
 }
 class ListenerQuitter implements ActionListener
 { public void actionPerformed(ActionEvent e)
   { System.exit(0);
   }
 }

 class AppliWindowEvent
 {
   public static void main(String [] arg) {
    Frame fen = new Frame ("Bonjour" );
    fen.setBounds(100,100,250,120);
    fen.setLayout(new FlowLayout( ));
    Button entree = new Button("Entrez");
    Button quitter = new Button("Quitter l'application");
    Label texte = new Label("Cette ligne est du texte");
    entree.addActionListener(new ListenerGeneral( entree, texte, fen ));
    quitter.addActionListener(new ListenerQuitter( ));
    fen.add(texte);
    fen.add(entree);
    fen.add(quitter);
    fen.setVisible(true);
    }
 }
```

Voici ce que devient l'interface après un click du bouton "Entrez" :



Intérêt d'hériter d'une classe XXXAdapter

Fermer une fenêtre directement sans passer par un bouton

Nous voulons pour terminer les exemples et utiliser un autre composant que le Button, fermer une fenêtre classiquement en cliquant sur l'icône du bouton de fermeture situé dans la barre de titre de la fenêtre et donc arrêter l'application. La démarche que nous adoptons est semblable à celle que nous avons tenue pour le click de bouton.

La documentation Java nous précise que l'interface des écouteurs qui ont trait aux événements de **bas niveau** des fenêtres, se dénomme WindowListener (équivalente à MouseListener). Les événements de **bas niveau** sont des objets instanciés à partir de la classe **java.awt.event.WindowEvent** qui décrivent les différents états d'une fenêtre

Il existe une classe implémentant l'interface WindowListener qui permet d'instancier des écouteurs d'actions sur les fenêtres, c'est la classe des WindowAdapter (à rapprocher de la classe déjà vue MouseAdapter). Dans ce cas, comme précédemment, il suffit de redéfinir la méthode qui est chargée d'intercepter et de traiter l'événement de classe **WindowEvent** qui nous intéresse.

Méthode à redéfinir	Action déclenchant l'événement
void windowActivated(WindowEvent e)	invoquée lorsqu'une fenêtre est activée.
void windowClosed(WindowEvent e)	invoquée lorsqu'une fenêtre a été fermée.
void windowClosing(WindowEvent e)	invoquée lorsqu'une fenêtre va être fermée.
void windowDeactivated(WindowEvent e)	invoquée lorsqu'une fenêtre est désactivée.
void windowDeiconified(WindowEvent e)	invoquée lorsqu'une fenêtre est sortie de la barre des tâches.
void windowIconified(WindowEvent e)	invoquée lorsqu'une fenêtre est mise en icône dans la barre des tâches.

void windowOpened(WindowEvent e)	invoquée lorsqu'une fenêtre est ouverte.

Dans notre cas c'est la méthode **void** windowClosing(WindowEvent e) qui nous intéresse, puisque nous souhaitons terminer l'application à la demande de fermeture de la fenêtre.

Nous écrivons le code le plus court : celui associé à une classe anonyme .

Version avec une classe anonyme

```
import java.awt.*;
import java.awt.event.*;

 class ApplicationCloseWin
 {
    public static void main(String [ ] arg)
    {
       Frame fen = new Frame ("Bonjour" );
       fen.addWindowListener (new WindowAdapter( )   {
          public void windowClosing(WindowEvent e)
         { System.exit(0);
          }
       });
      fen.setBounds(100,100,250,150);
      fen.setVisible(true);
    }
 }
```

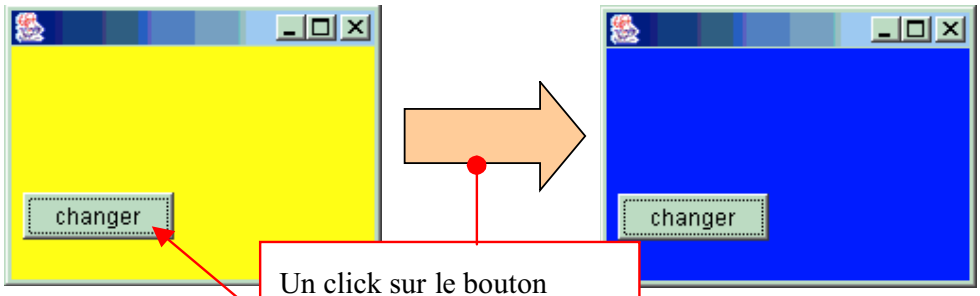
Affiche la fenêtre ci-dessous (les 3 boutons de la barre de titre fonctionnent comme une fenêtre classique, en particulier le dernier à droite ferme la fenêtre et arrête l'application lorsque l'on clique dessus) :

Exercices

Java2 IHM - Awt

Trois versions d'écouteur pour un changement de couleur du fond

Soit l'IHM suivante composée d'une fiche de classe **Frame** et d'un bouton de classe **Button** :




```
import  java.awt.* ;
import  java.awt.event.* ;

public class  ExoAwt0   {
 Frame fen  =  new  Frame ( );

 class  ecouteur  extends  MouseAdapter    {

  public void  mouseClicked ( MouseEvent e )    {
   fen.setBackground ( Color.blue );
  }
 }

 public  ExoAwt0 ( )   {
  fen.setBounds ( 50,50,200,150 );
  fen.setLayout (null);
  fen.setBackground ( Color.yellow );
  Button bouton  =  new  Button ("changer");
  ecouteur Bigears  =  new  ecouteur ( );
  bouton.addMouseListener ( Bigears );
  bouton.setBounds ( 10,100,80,25 );
  fen.add ( bouton );
  fen.setVisible (true) ;
 }

 public static void  main ( String [ ] x )    {
  new  ExoAwt0 ( );
 }
}
```

Première version avec une classe interne d'écouteur dérivant des MouseAdapter.

Instanciation de l'objet écouteur, puis recensement auprès du bouton.

```
import  java.awt. * ;
import  java.awt.event. * ;

class  ecouteur  extends  MouseAdapter   {
 private  Fenetre fenLocal ;

 public  ecouteur ( Fenetre F )   {
  fenLocal = F ;
 }

 public void  mouseClicked ( MouseEvent e )   {
  fenLocal.setBackground ( Color.blue );
 }
}

class  Fenetre  extends  Frame   {

 public  Fenetre ( )   {
  this.setBounds ( 50,50,200,150 );
  this.setLayout (null);
  this.setBackground ( Color.yellow );
   Button bouton  =  new  Button ("changer");
   ecouteur Bigears  =  new  ecouteur (this);
   bouton.addMouseListener ( Bigears );
   bouton.setBounds ( 10,100,80,25 );
  this.add ( bouton );
  this.setVisible ( );
 }
}

public class  ExoAwt   {

 public static void  main ( String [ ] x )   {
  Fenetre fen  =  new  Fenetre ( );
 }
}
```

Deuxième version avec une classe externe d'écouteur dérivant des MouseAdapter.

Instanciation de l'objet écouteur, puis recensement auprès du bouton.

Lors de la construction de l'écouteur **Bigears** la référence de la fiche elle-même **this**, est passée comme paramètre au constructeur.

Le champ local **fenLocal** reçoit cette référence et pointe vers la fiche,ce qui permet à l'écouteur d'accéder à tous les membres public de la fiche.

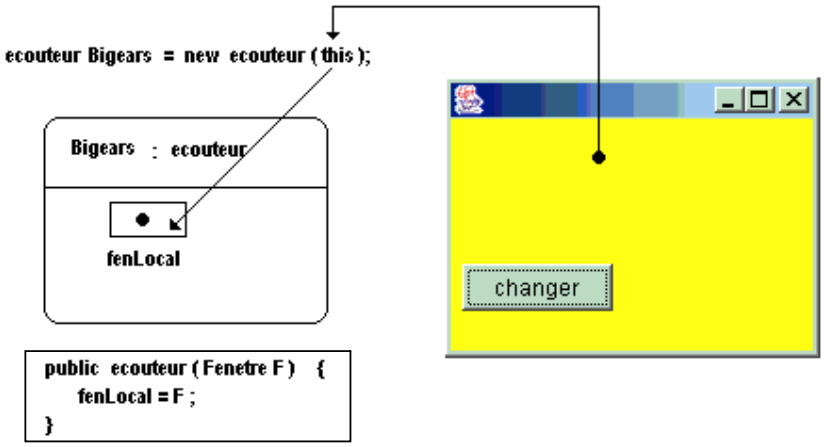


fig - schéma d'accès à la fiche par l'écouteur de classe externe

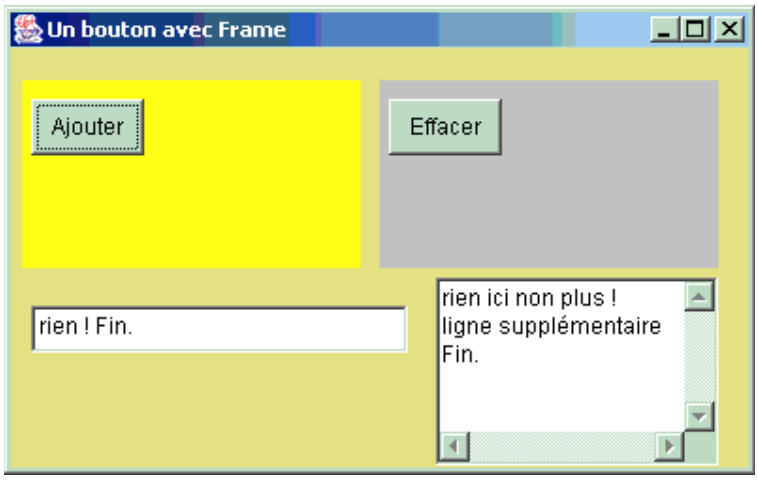
Voici la version la plus courte en code, version conseillée lorsque l'on n'a pas de travail particulier à faire exécuter par un écouteur et que l'on n'a pas besoin d'utiliser la référence de cet écouteur. Cette version utilise la notion de classe anonyme qui est manifestement très adaptée aux écouteurs :

```
import  java.awt. * ;
import  java.awt.event. * ;

class  Fenetre  extends  Frame    {

 void  GestionnaireClick ( MouseEvent e )   {
  this .setBackground ( Color.blue );
 }

 public  Fenetre ( )   {
  this.setBounds ( 50,50,200,150 );
  this.setLayout (null);
  this.setBackground ( Color.yellow );
   Button bouton  =  new  Button ("changer");
   bouton.addMouseListener (  new  MouseAdapter ( )    {
                                 public void  mouseClicked ( MouseEvent e )   {
                                     GestionnaireClick ( e );
                                  }

                                 }
                            ) ;
    bouton.setBounds ( 10,100,80,25 );
    this.add ( bouton );
    this.setVisible ( );
  }
}

public class  ExoAwtAnonyme   {

   public static void  main ( String [ ] x )   {
     Fenetre fen  =  new  Fenetre ( );
   }
}
```

Troisième version avec une classe anonyme d'écouteur dérivant des MouseAdapter.

IHM - Awt : Evénements de Button et TextField, stockage dans un TextArea sur un fenêtre qui se ferme : solution détaillée

Soit l'IHM suivante composée d'une fiche de classe **Frame**, de deux bouton Bout1 et Bout2 de classe **Button** déposés chacun sur un paneau (Panel1 et Panel2) de classe **Panel**, d'un éditeur de texte mono-ligne Edit1 de classe **TextField** et d'un éditeur de texte multi-ligne Memo1 de classe **TextArea**.



Nous définissons un certain nombre d'événements et nous les traitons avec le code le plus court lorsque cela est possible, soit avec des classes anonymes

Evénément-1 : La fiche se ferme et l'application s'arrête dès que l'utilisateur clique dans le bouton de fermeture de la barre de titre de la fenêtre.

La classe abstraite de gestion des événements de fenêtre se dénomme **WindowAdapter** et propose 10 méthodes vides à redéfinir dans un écouteur, chacune gérant un événement de fenêtre particulier. Chaque méthode est appelée lorsque l'evénement qu'elle gère est lancé :

void windowActivated(**WindowEvent** e) = appelée lorsque la fenêtre est activée.
void windowClosed(**WindowEvent** e) = appelée lorsque la fenêtre vient d'être fermée.
void windowClosing(**WindowEvent** e) = appelée lorsque la fenêtre va être fermée.
Etc…

Le paramètre **WindowEvent** e est l'objet d'événement que la fenêtre transmet à l'écouteur (ici c'est un événement de type **WindowEvent**)

Nous choisissons d'intercepter le windowClosing et de lancer la méthode **exit** de la classe System pour arrêter l'application :

Classe anonyme d'écouteur dérivant des WindowAdapter**.**

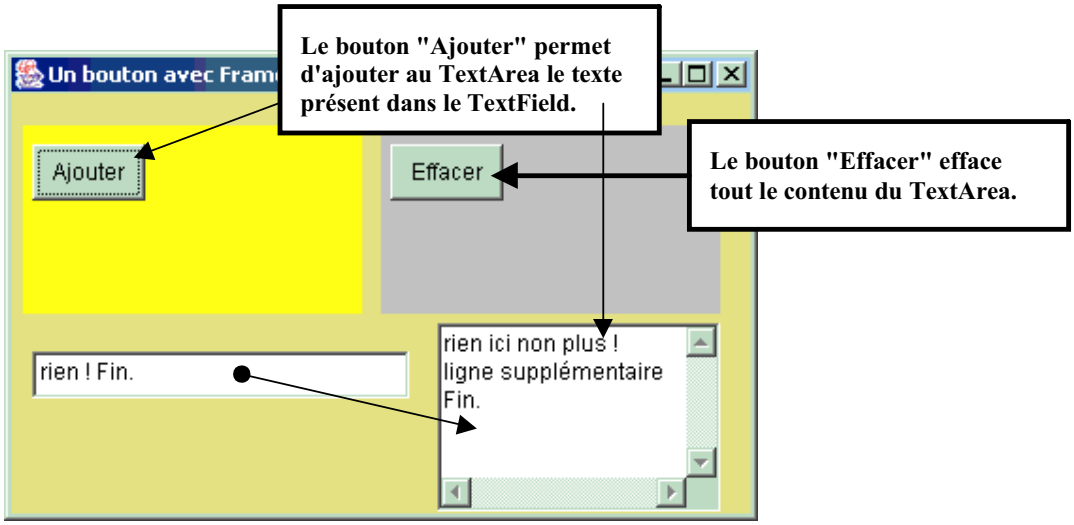
```
this.addWindowListener (   new  WindowAdapter ( )
   {
        public void  windowClosing ( WindowEvent e )   {
          System.exit ( 0 );
        }
   }
 );
```

L'écouteur gère le windowClosing**.**

Evénéments-2 :

- Le bouton Bout1 lorsque l'on clique sur lui, ajoute dans l'éditeur Memo1 la ligne de texte contenue dans l'éditeur Edit.

- Le bouton Bout2 lorsque l'on clique sur lui,efface le texte de Memo1.



La classe abstraite de gestion des événements de souris se dénomme **MouseAdapter** et propose 5 méthodes vides à redéfinir dans un écouteur, chacune gérant un événement de souris particulier. Chaque méthode est appelée lorsque l'evénement qu'elle gère est lancé :

void mouseClicked (**MouseEvent** e) = appelée lorsque l'on vient de cliquer avec la souris
Etc…

Le paramètre **MouseEvent** e est l'objet d'événement que le bouton transmet à l'écouteur (ici c'est un événement de type **MouseEvent**)

Nous choisissons d'intercepter le mouseClicked pour les deux boutons Bout1 et Bout2 :

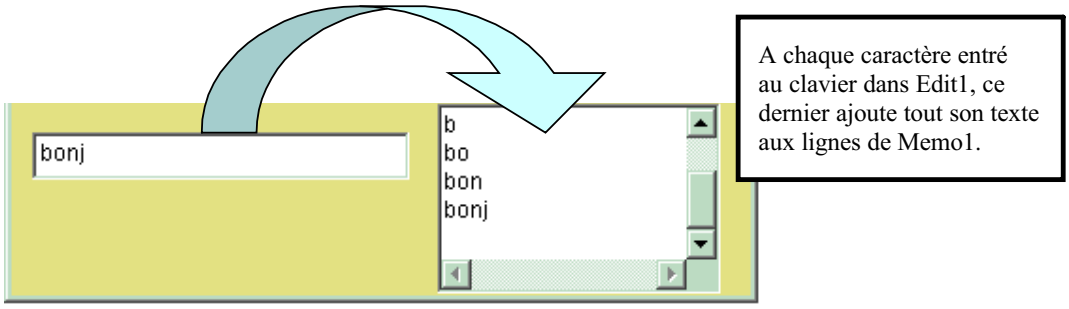
Classe anonyme d'écouteur dérivant des MouseAdapter**.**

```
Bout1.addMouseListener (   new  MouseAdapter ( )
 {
  public void  mouseClicked ( MouseEvent e )  {
   if( Edit1.getText () .length ()  !=  0 )
     Memo1.append ( Edit1.getText () + "/" + Edit1.getText ( ) .length () + "\n");
  }
 }
 );
  Bout2.addMouseListener (   new  MouseAdapter ( )
 {
  public void  mouseClicked ( MouseEvent e )  {
    Memo1.setText (null);
  }
 }
 );
```

L'écouteur gère le mouseClicked

Evénément-3 :

Lorsque le texte de l'Edit1 change, la ligne de texte contenue dans l'éditeur Edit s'ajoute dans le Memo1 :



L'interface de gestion des événements de souris se dénomme **TextListener** et propose une seule méthode vide à redéfinir dans un écouteur. C'est pourquoi il n'y a pas de classe abstraite du genre **TextAdapter** car il suffit que la classe d'écouteur implémente l'interface (au lieu d'hériter de la classe xxxAdapter) et redéfinisse la seule méthode de l'interface :

void textValueChanged(**TextEvent** e) = appelée lorsque la valeur du texte a changé.

Le paramètre **TextEvent** e est l'objet d'événement que le bouton transmet à l'écouteur (ici c'est un événement de type **TextEvent**)

```
Edit1.addTextListener ( new TextListener ( )
 {
  public void textValueChanged ( TextEvent e ) {
    Memo1.append ( Edit1.getText ( ) + "\n");
  }
 }
);
```

Classe anonyme d'écouteur implémentant TextListener**.**

L'écouteur gère le textValueChanged

```
/*
      Une Fenêtre avec 2 panels avec bouton,  un TextField et un TextArea.
      avec interception d'événements par classe anonyme :
       code le plus court possible !
*/
import java.awt. * ;
import java.awt.event. * ;

public class FrameBasic extends Frame {
  Button Bout1 = new Button ("Ajouter");
  Button Bout2 = new Button ("Effacer");
  Panel Panel1 = new Panel ();
  //si Panel2 = new Panel() => alors FlowLayout manager par défaut :
  Panel Panel2 = new Panel (null);
  TextField Edit1 = new TextField ("rien !");
  TextArea Memo1 = new TextArea ("rien ici non plus !");

 public FrameBasic ( ) {
  this.setBounds ( 80,100,400,250 );
  this.setTitle ("Un bouton avec Frame");
```

```
this.setBackground ( Color.orange );

 Panel1.setBounds ( 10,40,180,100 );
 Panel1.setBackground ( Color.red );
 Panel1.setLayout (null);

 Panel2.setBounds ( 200,40,180,100 );
 Panel2.setBackground ( Color.blue );
 //Panel2.setLayout(new BorderLayout());

 Bout1.setBounds ( 5, 10, 60, 30 );
 Bout2.setBounds ( 5, 10, 60, 30 );
 Edit1.setBounds ( 15, 160, 200, 25 );
 Edit1.setText ( Edit1.getText ( ) + " Fin.");
 Memo1.setBounds ( 230, 145, 150, 100 );
 Memo1.append ("\n");
 Memo1.append ("ligne supplémentaire\n");
 Memo1.append ("Fin.\n");

 Panel1.add ( Bout1 );
 Panel2.add ( Bout2 );

this.setLayout (null);
this.add ( Panel1 );
this.add ( Panel2 );
this.add ( Edit1 );
this.add ( Memo1 );
this.setVisible ( true );

this.addWindowListener (  new  WindowAdapter ( )
{
 public void  windowClosing ( WindowEvent e )  {
   System.exit ( 0 );
 }
}
);
 Bout1.addMouseListener (  new  MouseAdapter ( )
{
 public void  mouseClicked ( MouseEvent e )  {
  if( Edit1.getText ( ) .length ( )  !=  0 )
    Memo1.append ( Edit1.getText ( ) + "/" + Edit1.getText ) .length ( ) + "\n");
 }
}
);
 Bout2.addMouseListener (  new  MouseAdapter ( )
{
 public void  mouseClicked ( MouseEvent e )  {
   Memo1.setText (null);
 }
}
);
 Edit1.addTextListener (  new  TextListener ( )
{
 public void  textValueChanged ( TextEvent e )  {
   Memo1.append ( Edit1.getText ( ) + "\n");
 }
}
);
}
}
```

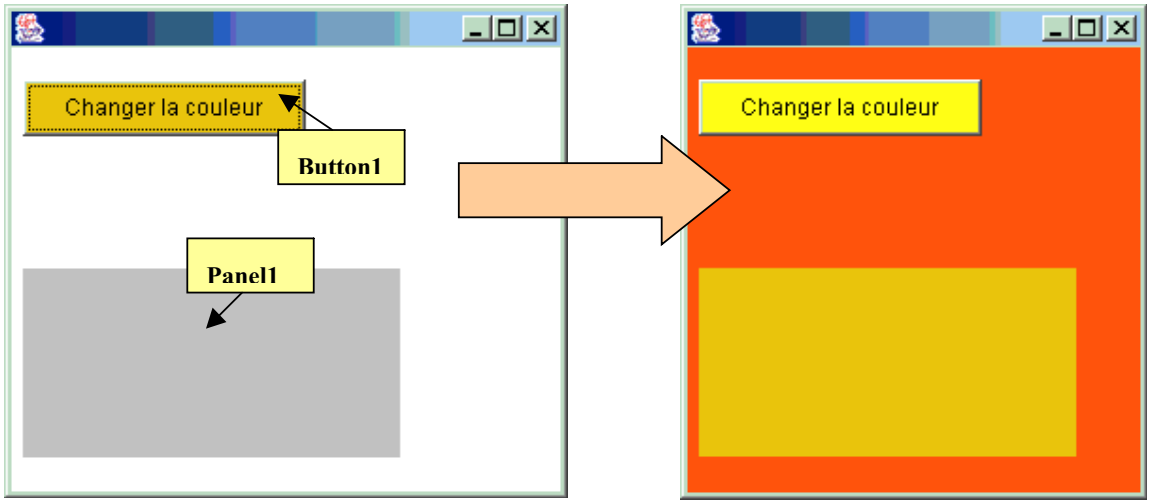
IHM - Awt : Variations de souris sur une fenêtre et écouteur centralisé

Deux versions d'une même IHM

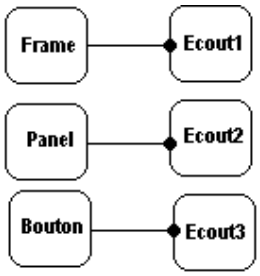
Soit l'IHM suivante composée d'une fiche de classe **Frame**, d'un bouton Button1 de classe Bouton dérivée de la classe Button , et d'un panneau Panel1 de classe **Panel**.

L'IHM réagit uniquement au click de souris :

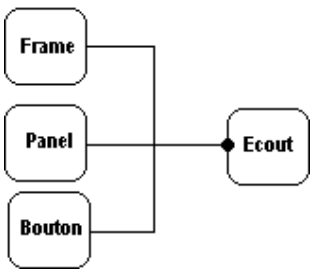
- Le Button1 de classe Bouton réagit au simple click et il fait alternativement changer de couleur le fond de la fiche sur laquelle il est déposé.
- Le Panel1 de classe Panel réagit au simple click et au double click, chaque réaction click ou double-click fait changer sa couleur de fond.
- La fiche de classe Frame est sensible au click de souris pour sa fermeture, au click de souris sur son fond et au double click sur son fond (chaque réaction click ou double-click fait changer sa couleur de fond).



Si nous choisissons d'utiliser un écouteur de classe héritant des WindowAdapter.



Nous pouvons instancier pour chaque objet (fenêtre, panneau et bouton) un écouteur qui doit redéfinir la méthode mouseClicked



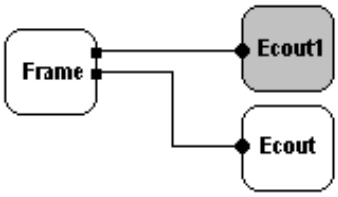
Nous pouvons aussi instancier un écouteur général (**centralisé**) qui écoutera les 3 objets.

```
/*
Une Fenêtre où l'on intercepte les événements de click de souris en  utilisant un écouteur d'événements
WindowAdapter  pour la fenêtre et un écouteur d'événements MouseAdapter pour: fenêtre, panneau et bouton.
*/

import  java.awt. * ;
import  java.awt.event. * ;

class  AppliUneFrameClick2
{

 //-- classe interne écouteur centralisé de souris MouseAdapter :
 class  SourisAdapter  extends  MouseAdapter
 {
  //écoute les événements de click de souris de la fenêtre, du panneau et du bouton !
  public void  mouseClicked ( MouseEvent e )  {
  if( e.getSource( )  instanceof  Fenetre )   {
    Fenetre FicheaEcouter  =  ( Fenetre )( e.getSource ( ));
   if( e.getClickCount ( ) == 1 )
     FicheaEcouter.GestionMouseClicked ( e );
   else
     FicheaEcouter.GestionMouseDblClicked ( e );
  }
  else   {
   if( e.getSource ( )  instanceof  Panneau )    {
     Panneau PanneauaEcouter  =  ( Panneau )( e.getSource ( ) );
    if( e.getClickCount ( ) == 1 )
      PanneauaEcouter.GestionMouseClicked ( e );
    else
      PanneauaEcouter.GestionMouseDblClicked ( e );
   }
   else   {
    if( e.getSource ( )  instanceof  Bouton )    {
      Bouton ButtonaEcouter  =  ( Bouton )( e.getSource ( ));
      ButtonaEcouter.GestionMouseClicked ( e );
    }
   }
  }
 }
}
 //-- classe interne écouteur de fenêtre WindowAdapter :
 class  FenetreAdapter  extends  WindowAdapter
 {
  Fenetre FicheaEcouter  ;

  FenetreAdapter ( Fenetre x )   {
   FicheaEcouter  =  x ;
  }
  public void  windowClosing ( WindowEvent e )  {
   FicheaEcouter.GestionWindowClosing ( e );
  }
 }
 //-- classe interne un Panneau dans la fenêtre :
 class  Panneau  extends  Panel
 {
 public  Panneau ( Frame AOwner )  {
   AOwner.add (this);
  this.setBounds ( 10,140,200,100 );
```

Si l'émetteur du MouseEvent est du type Fenetre.

Si l'émetteur du MouseEvent est du type Panneau.

Si l'émetteur du MouseEvent est du type Bouton.



La fenêtre possède deux écouteurs :

- **L'un pour sa fermeture (** WindowAdapter **)**
- **L'autre centralisé pour les click sur son fond (** MouseAdapter **)**

```
 this.setBackground ( Color.lightGray );
 }
 void  GestionMouseClicked ( MouseEvent e )  {
  this .setBackground ( Color.magenta );
 }
 void  GestionMouseDblClicked ( MouseEvent e )  {
  this .setBackground ( Color.orange );
 }

}
 //-- classe interne un Button dans la fenêtre :
 class  Bouton  extends  Button
 {
 public  Bouton ( Frame AOwner )  {
   AOwner.add (this);
  this.setBounds ( 10,40,150,30 );
  this.setLabel ("Changer la couleur");
  this.setBackground ( Color.orange );
 }
 void  GestionMouseClicked ( MouseEvent e )  {
  if (this.getBackground () == Color.yellow )  {
   this.setBackground ( Color.cyan );
   this.getParent ( ).setBackground ( Color.green );
  }
  else  {
   this.setBackground ( Color.yellow );
   this.getParent ( ).setBackground ( Color.red );
  }
 }
}
 //-- classe interne une fenêtre dans l'application :
 class  Fenetre  extends  Frame
 {
  SourisAdapter UnEcouteurSourisEvent  =  new  SourisAdapter ( );
  FenetreAdapter UnEcouteurFenetreEvent  =  new  FenetreAdapter (this);
  Panneau panel1  =  new  Panneau (this);
  Bouton Button1  =  new  Bouton (this);

 public  Fenetre ( )  {
  this.setLayout (null);
  this.setSize (new  Dimension ( 400, 300 ));
  this.setTitle ("MouseAdapter dans la fenêtre,le panneau et le bouton");
  this.setVisible ( true );
  Button1.addMouseListener ( UnEcouteurSourisEvent );
  panel1.addMouseListener ( UnEcouteurSourisEvent );
  this.addMouseListener ( UnEcouteurSourisEvent );
  this.addWindowListener ( UnEcouteurFenetreEvent );
 }

 void  GestionWindowClosing ( WindowEvent  e )  {
   System.exit ( 0 );
 }

 void  GestionMouseClicked ( MouseEvent e )  {
  this.setBackground ( Color.blue );
 }

 void  GestionMouseDblClicked ( MouseEvent e )  {
  this.setBackground ( Color.pink );
 }
```

this.getParent ( ) renvoie une référence sur le parent de l'objet Bouton : dans l'exercice le parent est la fiche

La fiche héritant de Frame avec ses composants déposés.

L'écouteur centralisé est recensé auprès des 3 objets.

La fiche this recense son écouteur pour WindowClosing

```
}
 //---> constructeur de l'application :
 AppliUneFrameClick2 ( ) {
  Fenetre Fiche2  =  new  Fenetre ( );
}

public static void  main ( String [ ] args )  {
 new  AppliUneFrameClick2 ( );
}
}
```

Pour la classe Bouton, on peut aussi déclarer un champ privé du type Frame (**private** Frame fenLoc) qui stocke la référence de la fiche contenant le Bouton. C'est le constructeur de Bouton qui passe alors la référence effective de la fenêtre.

Nous remarquons dans le code ci-dessous à droite que le fait de disposer de la référence (**private** Frame fenLoc) sur la fiche qui contient le bouton offre plus de possibilités que le code de gauche où il a fallu faire appel au parent par la méthode getParent pour accéder à la fiche :

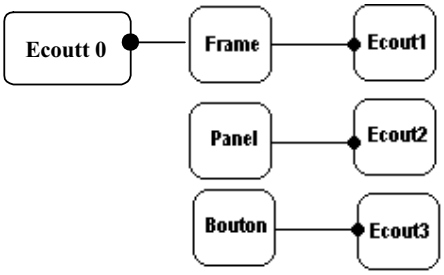
Accès à la fiche comme parent

```
//-- classe interne un Button dans la fenêtre :
class  Bouton  extends  Button
{
public  Bouton ( Frame AOwner )  {
  AOwner.add (this);
 this.setBounds ( 10,40,150,30 );
 this.setLabel ("Changer la couleur");
 this.setBackground ( Color.orange );
}
void  GestionMouseClicked ( MouseEvent e )  {
 if (this.getBackground () == Color.yellow )  {
  this.setBackground ( Color.cyan );
  this.getParent ( ).setBackground ( Color.green );
 }
 else  {
  this.setBackground ( Color.yellow );
  this.getParent ( ).setBackground ( Color.red );
 }
}
}
```

Accès à la fiche par une référence

```
//-- classe interne un Button dans la fenêtre :
class  Bouton  extends  Button
{

private Frame FenLoc;
public  Bouton ( Frame AOwner )  {
  AOwner.add (this); FenLoc = Aowner ;
 this.setBounds ( 10,40,150,30 );
 this.setLabel ("Changer la couleur");
 this.setBackground ( Color.orange );
}
void  GestionMouseClicked ( MouseEvent e )  {
 if (this.getBackground ( ) == Color.yellow )  {
  this.setBackground ( Color.cyan );
  FenLoc.setBackground ( Color.green );
 }
 else  {
  this.setBackground ( Color.yellow );
  FenLoc.setBackground ( Color.red );
 }
}
}
```

Voici maintenant la deuxième version de codage proposée pour l'IHM précédente en utilisant pour tous les écouteurs une classe anonyme :




```
/*
Une Fenêtre où l'on intercepte les événements de click de souris en utilisant un écouteur d ' événements    fenêtre et
un écouteur d ' événements souris avec des classes anonymes  !
*/

import  java.awt. * ;
import  java.awt.event. * ;

class  AppliUneFrameClick3
{

 //-- classe interne un Panneau dans la fenêtre :
 class  Panneau  extends  Panel
 {
   … code strictement identique à la version précédente ….
 }
 //-- classe interne un Button dans la fenêtre :
 class  Bouton  extends  Button
 {
   … code strictement identique à la version précédente ….
 }


 //-- classe interne une fenêtre dans l'application :
 class  Fenetre  extends  Frame
 {
  Panneau panel1  =  new  Panneau (this);
  Bouton Button1  =  new  Bouton (this);

 public  Fenetre ( )  {
  this.setLayout (null);
  this.setSize ( new  Dimension ( 400, 300 ) );
  this.setTitle ("Classe anonyme pour la fenêtre,le panneau et le bouton");
  this.setVisible ( true );
   Button1.addMouseListener (  new  java.awt.event.MouseAdapter ( )
  {
   public void  mouseClicked ( MouseEvent e )    {
     Button1.GestionMouseClicked ( e );
   }
  }
  );
   panel1.addMouseListener (  new  java.awt.event.MouseAdapter ( )
  {
   public void  mouseClicked ( MouseEvent e )   {
    if( e.getClickCount ( ) == 1 )
      panel1.GestionMouseClicked ( e );
```

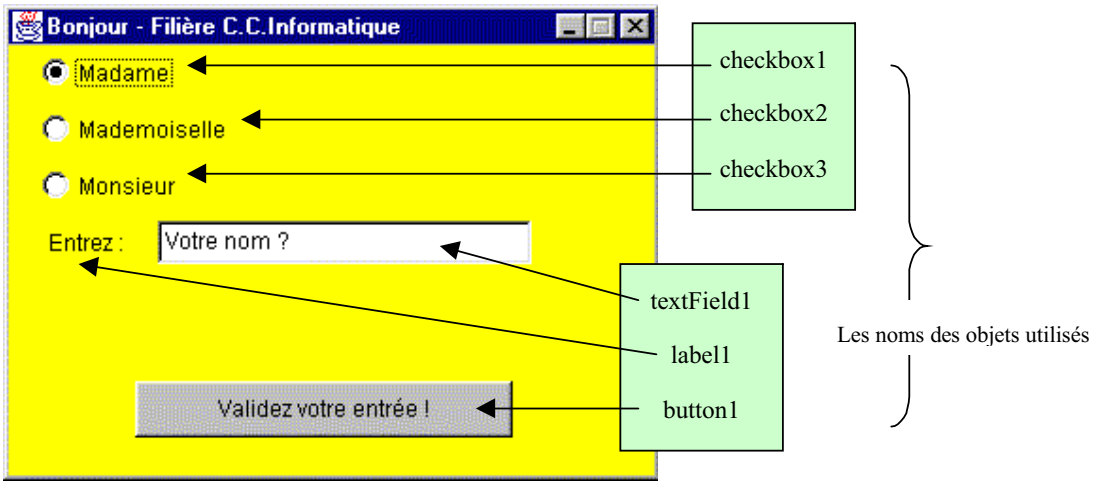
Classes anonymes héritant de MouseAdapter

```
    else
      panel1.GestionMouseDblClicked ( e );
   }
  }
  );
  this .addMouseListener (  new  java.awt.event.MouseAdapter ( )
  {
   public void  mouseClicked ( MouseEvent e )     {
    if( e.getClickCount () == 1 )
      Fenetre.this.GestionMouseClicked ( e );
    else
      Fenetre.this.GestionMouseDblClicked ( e );
   }
  }
  );
  this .addWindowListener (   new  java.awt.event.WindowAdapter ( )   {
   public void  windowClosing ( WindowEvent e )
   {
    Fenetre.this.GestionWindowClosing ( e );
   }
  }
  );
 }

 void  GestionWindowClosing ( WindowEvent  e )   {
   System.exit ( 0 );
 }

 void  GestionMouseClicked ( MouseEvent e )   {
  this .setBackground ( Color.blue );
 }

 void  GestionMouseDblClicked ( MouseEvent e )  {
  this .setBackground ( Color.pink );
 }
}

 AppliUneFrameClick3 ()
{
  Fenetre Fiche3  =  new  Fenetre ();
}

public static void  main ( String [ ] args )  {
 new  AppliUneFrameClick3 ( );
}
}
```

Identique au code de la version précédente

IHM - Awt : Saisie de renseignements interactive

Nous reprenons l'IHM de saisie de renseignements concernant un(e) étudiant(e) que nous avons déjà construite sans événement, rajoutons des événements pour la rendre interactive, elle stockera les renseignements saisis dans un fichier de texte éditable avec un quelconque traitement de texte :



Description événementielle de l'IHM :

- Dans l'IHM au départ le **button1** est désactivé, aucun **checkbox** n'est coché, le **textField1** est vide.
- Dès que l'un des **checkbox** est coché, et que le **textField1** contient du texte le **button1** est activé, dans le cas contraire le **button1** est désactivé (dès que le **textField1** est vide).
- Un click sur le **button1** sauvegarde les informations dans le fichier texte **etudiants.txt**.
- La fiche se ferme et arrête l'application sur le click du bouton de fermeture.

```
import java.awt.*; // utilisation des classes du package awt
  import java.awt.event.*; // utilisation des classes du package awt
  import java.io.*; // utilisation des classes du package io

   public class AppliSaisie { //  classe principale
  //Méthode principale
     public static void main(String[ ] args) { // lance le programme
       ficheSaisie fenetre = new ficheSaisie ( );// création d'un objet de classe ficheSaisie
       fenetre.setVisible(true);// cet objet de classe ficheSaisie est rendu visible sur l'écran
    }
  }

   class ficheSaisie extends Frame { // la classe Cadre1 hérite de la classe des fenêtres Frame
```

```
    Button button1 = new Button( );// création d'un objet de classe Button
    Label label1 = new Label( );// création d'un objet de classe Label
    CheckboxGroup checkboxGroup1 = new CheckboxGroup( );// création d'un objet groupe de checkbox
    Checkbox checkbox1 = new Checkbox( );// création d'un objet de classe Checkbox
    Checkbox checkbox2 = new Checkbox( );// création d'un objet de classe Checkbox
    Checkbox checkbox3 = new Checkbox( );// création d'un objet de classe Checkbox
    TextField textField1 = new TextField( );// création d'un objet de classe TextField

    private String EtatCivil;//champ = le label du checkbox coché
    private FileWriter fluxwrite; //flux en écriture (fichier texte)
    private  BufferedWriter fluxout;//tampon pour lignes du fichier

  //Constructeur de la fenêtre
     public ficheSaisie ( ) {  //Constructeur sans paramètre
       Initialiser( );// Appel à une méthode privée de la classe
    }
  //Active ou désactive le bouton pour sauvegarde :
     private void AutoriserSave(){
      if (textField1.getText().length() !=0 &&  checkboxGroup1.getSelectedCheckbox() != null)
        button1.setEnabled(true);
      else
        button1.setEnabled(false);
    }
  //rempli le champ Etatcivil selon le checkBox coché :
     private void StoreEtatcivil(){
      this.AutoriserSave();
      if (checkboxGroup1.getSelectedCheckbox() != null)
        this.EtatCivil=checkboxGroup1.getSelectedCheckbox().getLabel();
      else
        this.EtatCivil="";
    }
  //sauvegarde les infos étudiants dans le fichier :
     public void ecrireEnreg(String record) {
      try {
        fluxout.write(record);//écrit les infos
        fluxout.newLine( ); //écrit le eoln
      }
         catch (IOException err) {
          System.out.println( "Erreur : " + err );
        }
    }
  //Initialiser la fenêtre :
     private void Initialiser( ) {  //Création et positionnement de tous les composants
      this.setResizable(false); // la fenêtre ne peut pas être retaillée par l'utilisateur
      this.setLayout(null); // pas de Layout, nous positionnons les composants nous-mêmes
      this.setBackground(Color.yellow); // couleur du fond de la fenêtre
      this.setSize(348, 253); // widht et height de la fenêtre
      this.setTitle("Bonjour - Filière C.C.Informatique"); // titre de la fenêtre
      this.setForeground(Color.black); // couleur de premier plan de la fenêtre
      button1.setBounds(70, 200, 200, 30); // positionnement du bouton
      button1.setLabel("Validez votre entrée !"); // titre du bouton
      button1.setEnabled(false); // bouton désactivé
      label1.setBounds(24, 115, 50, 23); // positionnement de l'étiquette
      label1.setText("Entrez :"); // titre de l'étiquette
      checkbox1.setBounds(20, 25, 88, 23); // positionnement du CheckBox
      checkbox1.setCheckboxGroup(checkboxGroup1); // ce CheckBox est mis dans le groupe checkboxGroup1
      checkbox1.setLabel("Madame");// titre du CheckBox
      checkbox2.setBounds(20, 55, 108, 23);// positionnement du CheckBox
      checkbox2.setCheckboxGroup(checkboxGroup1);// ce CheckBox est mis dans le groupe checkboxGroup1
      checkbox2.setLabel("Mademoiselle");// titre du CheckBox
```

```
        checkbox3.setBounds(20, 85, 88, 23);// positionnement du CheckBox
        checkbox3.setCheckboxGroup(checkboxGroup1);// ce CheckBox est mis dans le groupe checkboxGroup1
        checkbox3.setLabel("Monsieur");// titre du CheckBox
        textField1.setBackground(Color.white);// couleur du fond de l'éditeur mono ligne
        textField1.setBounds(82, 115, 198, 23);// positionnement de l'éditeur mono ligne
        textField1.setText("Votre nom ?");// texte de départ de l'éditeur mono ligne
        this.add(checkbox1);// ajout dans la fenêtre du CheckBox
        this.add(checkbox2);// ajout dans la fenêtre du CheckBox
        this.add(checkbox3);// ajout dans la fenêtre du CheckBox
        this.add(button1);// ajout dans la fenêtre du bouton
        this.add(textField1);// ajout dans la fenêtre de l'éditeur mono ligne
        this.add(label1);// ajout dans la fenêtre de l'étiquette
        EtatCivil ="";// pas encore de valeur
        try{
          fluxwrite = new FileWriter("etudiants.txt",true);// création du fichier(en mode ajout)
          fluxout = new BufferedWriter(fluxwrite); //tampon de ligne associé
        }
        catch(IOException err){ System.out.println( "Problème dans l'ouverture du fichier ");}
        //--> événements et écouteurs :
        this.addWindowListener(
            new WindowAdapter(){
              public void windowClosing(WindowEvent e){
                try{
                  fluxout.close( ); //le fichier est fermé et le tampon vidé
                }
                catch(IOException err){ System.out.println( "Impossible de fermer le fichier ");}
                System.exit(100);
              }
            });
        textField1.addTextListener(  new TextListener(){
              public void textValueChanged(TextEvent e) {
                AutoriserSave( );          }
            });
        checkbox1.addItemListener(  new ItemListener(){
              public void itemStateChanged(ItemEvent e){
                StoreEtatcivil( );
              }
            });
        checkbox2.addItemListener(  new ItemListener(){
              public void itemStateChanged(ItemEvent e){
                StoreEtatcivil();
              }
            });
        checkbox3.addItemListener(
            new ItemListener(){
              public void itemStateChanged(ItemEvent e){
                StoreEtatcivil();
              }
            });__
        button1.addMouseListener( new MouseAdapter(){
              public void mouseClicked(MouseEvent e){
                ecrireEnreg(EtatCivil+":"+textField1.getText());
              }
            });
      }
    }
```

Le texte du textField1 a changé

Click dans l'un des checkBox

Click sur le button1 "valider…"

IHM - Awt : Fermer une Frame directement par processWindowEvent

Il existe en Java un autre moyen d'intercepter les événements de fenêtre (objet de classe **WindowEvent**) sans utiliser un écouteur.

ProcessWindowEvent :

La méthode protégée **processWindowEvent** de la classe **Window** dont héritent les Frame, assume le passage de l'événement aux écouteurs recensés s'il en existe, et elle assure aussi le traitement direct d'un événement quelconque de classe **WindowEvent** par la fenêtre elle-même.

protected void processWindowEvent(**WindowEvent** e)

Un événement de classe **WindowEvent** est par héritage un **AwtEvent** caractérisé essentiellement par une valeur numérique sous forme d'un champ static de type **int** qui défini l'événement qui est en cause.

public abstract class **AWTEvent**

static long ACTION_EVENT_MASK
static long ADJUSTMENT_EVENT_MASK
….

static long WINDOW_EVENT_MASK
static long WINDOW_FOCUS_EVENT_MASK
static long WINDOW_STATE_EVENT_MASK

Ci-dessous les 12 champs nouveaux apportés par la classe **WindowEvent** :

Class WindowEvent

static int WINDOW_ACTIVATED
static int WINDOW_CLOSED
static int WINDOW_CLOSING
static int WINDOW_DEACTIVATED
static int WINDOW_DEICONIFIED
static int WINDOW_FIRST
static int WINDOW_GAINED_FOCUS
static int WINDOW_ICONIFIED
static int WINDOW_LAST
static int WINDOW_LOST_FOCUS
static int WINDOW_OPENED
static int WINDOW_STATE_CHANGED

Tout objet d'événement **evt** est un objet de classe **AwtEvent** et donc possède une méthode getID qui permet de connaître le type numérique de l'événement en le renvoyant comme résultat :

public int getID ( )

Dans le cas où **evt** est un **WindowEvent** dérivant des **AwtEvent,** les valeurs possibles de résultat de getID sont :

> **WindowEvent**.WINDOW_ACTIVATED,
>
> **WindowEvent**.WINDOW_CLOSED,
>
> **WindowEvent**.WINDOW_CLOSING,
>
> …etc

La méthode protégée **processWindowEvent** de la classe **Window** est appelée systématiquement par une fenêtre dès qu'un événement se produit sur elle, cette méthode envoie aux écouteurs recensés auprès de la fenêtre, l'événement qui lui est passé comme paramètre, mais peut donc traiter directement sans l'envoi de l'objet d'événement à un écouteur :

```
protected void processWindowEvent(WindowEvent e) {
   if (e.getID() == WindowEvent. WINDOW_ACTIVATED) {… traitement1 …. }
    else if (e.getID() == WindowEvent. WINDOW_CLOSED) {… traitement2 …. }
     else if (e.getID() == WindowEvent.WINDOW_CLOSING) {… traitement 3…. }
       etc…
}
```

Enfin, si nous programmons le corps de la méthode **processWindowEvent** pour un événement **evt**, comme nous venons de le faire, nous remplaçons le processus automatique d'interception par le nôtre, nous empêchons toute action autre que la nôtre. Or ce n'est pas exactement ce que nous voulons, nous souhaitons que notre fenêtre réagisse automatiquement et en plus qu'elle rajoute notre réaction à l'événement **evt**; nous devons donc d'abord hériter du comportement de la Frame (appel à la méthode **processWindowEvent** de la super-classe) puis ajouter notre code :

```
class Fenetre extends Frame {

 protected void processWindowEvent(WindowEvent e) {
  super.processWindowEvent(e);
  if (e.getID() == WindowEvent. WINDOW_ACTIVATED) {… traitement1 …. }
   else if (e.getID() == WindowEvent. WINDOW_CLOSED) {… traitement2 …. }
    else if (e.getID() == WindowEvent.WINDOW_CLOSING) {… traitement 3…. }
      etc…
 }
}
```

Pour que la méthode processWindowEvent agisse effectivement Java demande qu'une autorisation de filtrage du type de l'événement soit mise en place. C'est la méthode **enableEvents** qui se charge de fournir cette autorisation en recevant comme paramètre le masque (valeur numérique sous forme de champ static long du type de l'événement) :

```
protected final void enableEvents(long eventsToEnable)
```

Voici différents appels de **enableEvents** avec des masques différents

```
enableEvents ( AWTEvent. ACTION_EVENT_MASK);
enableEvents ( AWTEvent. ADJUSTMENT_EVENT_MASK);
enableEvents ( AWTEvent.WINDOW_EVENT_MASK ); … etc
```

Ce qui donne le code définitif de gestion directe d'un événement **WindowEvent** par notre classe de fenêtre (la propagation de l'événement s'effectue par appel de processWindowEvent de la classe mère avec comme paramètre effectif l'événement lui-même) :

```
class Fenetre extends Frame {
 public Fenetre( ) {
     enableEvents ( AWTEvent.WINDOW_EVENT_MASK );
}
protected void processWindowEvent(WindowEvent e) {
  super.processWindowEvent(e);
 if (e.getID() == WindowEvent.WINDOW_ACTIVATED) {… traitement1 …. }
  else if (e.getID() == WindowEvent.WINDOW_CLOSED) {… traitement2 …. }
    else if (e.getID() == WindowEvent.WINDOW_CLOSING) {… traitement 3…. }
       etc…
 }
}
```

Donc la fermeture d'une fenêtre héritant d'une Frame sur windowClosing peut se faire de deux façons :

Avec traitement direct	Avec un écouteur (anonyme ici)
`class Fenetre extends Frame {` `public Fenetre( ) {` `enableEvents ( AWTEvent.WINDOW_EVENT_MASK );` `}` `protected void processWindowEvent(WindowEvent e) {` `super.processWindowEvent(e);` `if (e.getID() == WindowEvent.WINDOW_CLOSING)` `System.exit(100);` `}` `}`	`class Fenetre extends Frame {` `public Fenetre( ) {` `this.addWindowListener( new WindowAdapter( ){` `public void windowClosing(WindowEvent e){` `System.exit(100);` `}` `});` `}`

Figure : traitement direct de la fermeture ou de la mise en icônes en barre des tâches

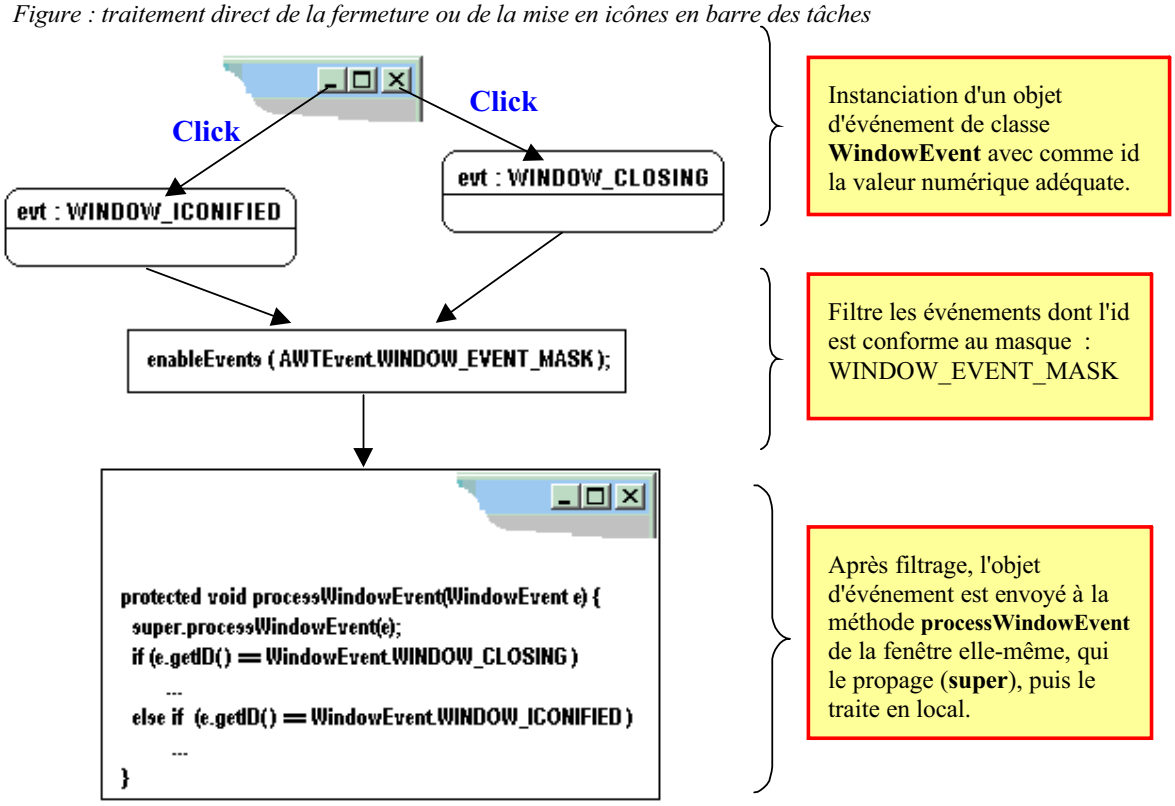
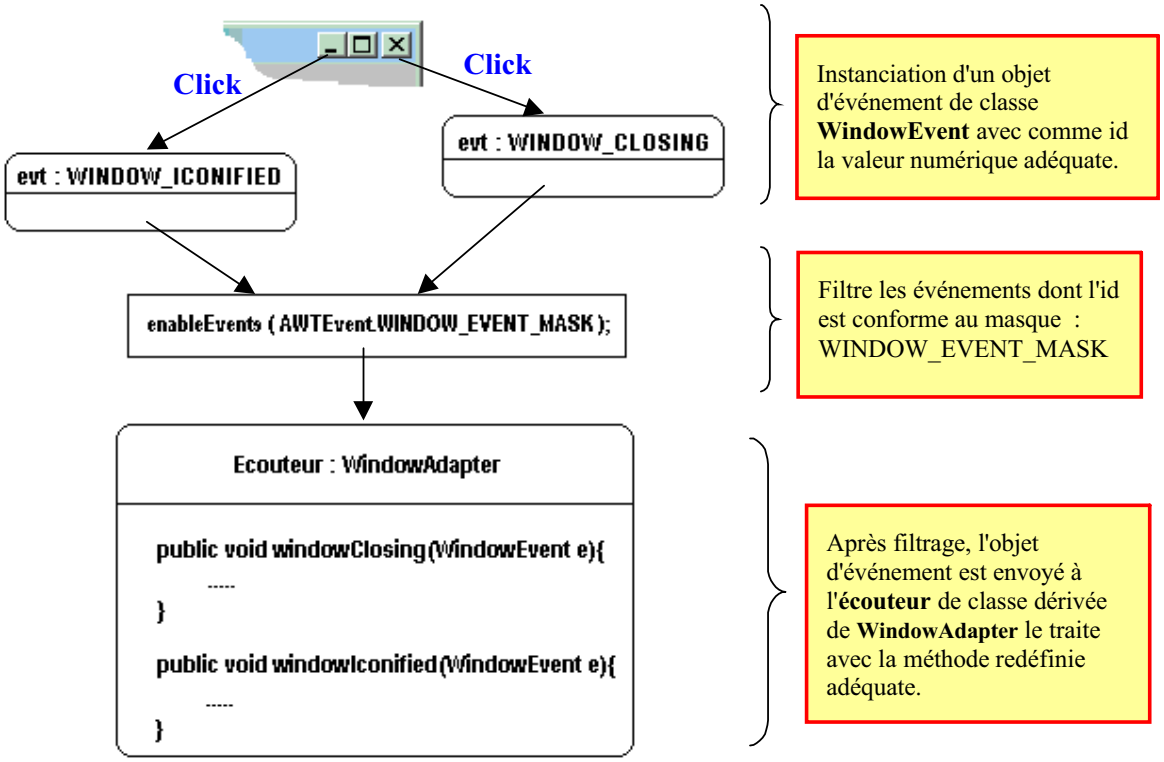


Figure : traitement par écouteur de la fermeture ou de la mise en icônes en barre des tâches

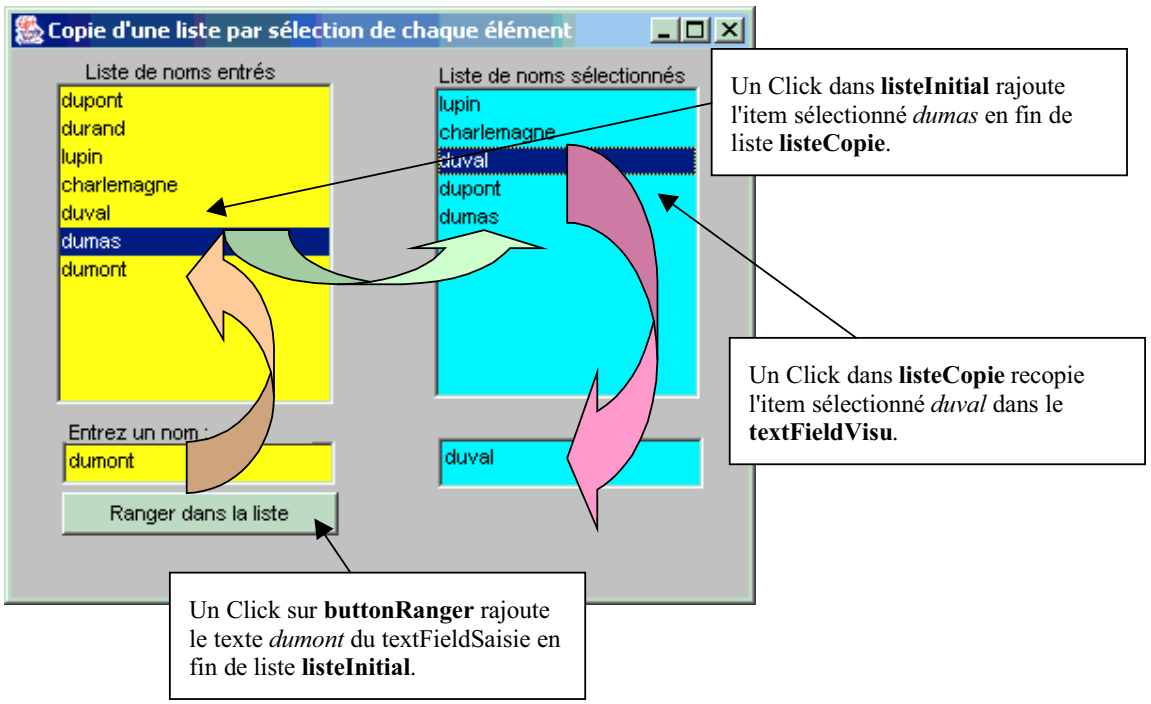


IHM - Awt : Utiliser une java.awt.List

Soit l'IHM suivante composée d'une fiche **Fenetre** de classe **Frame**, d'un bouton **buttonRanger** de classe **Button**, de deux composants de classe **textField** dénommés **textFieldSaisie** à gauche et **textFieldVisu** à droite, puis de deux objets de classe **List** dénommés **listeInitial** à gauche et **listCopie** à droite

L'IHM réagit uniquement au click de souris :

- Le **buttonRanger** de classe **Button** réagit au simple click et ajoute à la fin de la liste de gauche (objet **listeInitial**), le texte entré dans le **textFieldSaisie** à condition que ce dernier ne soit pas vide.
- Le **listeInitial** de classe **List** réagit au simple click du bouton gauche de souris et au double click de n'importe quel bouton de souris sur un élément sélectionné. Le simple click ajoute l'item sélectionné dans la liste de droite **listCopie**, le double click efface l'élément sélectionné de la liste **listeInitial.**
- Le **listCopie** de classe **List** réagit au simple click du bouton gauche de souris, il recopie l'item sélectionné par ce click dans le **textFieldVisu** en bas à droite .



Code Java de la classe Fenetre

```
import java.awt.*;
import java.awt.event.*;

public class Fenetre extends Frame {
    List listInitial = new List( );
    List listCopie = new List( );
    TextField textFieldSaisie = new TextField( );
    Button buttonRanger = new Button( );
```

```
TextField textFieldVisu = new TextField( );
Label label1 = new Label( );
Label label2 = new Label( );
Label label3 = new Label( ) ;

public Fenetre( ) {
 enableEvents( AWTEvent.WINDOW_EVENT_MASK );
 this.setBackground(Color.lightGray);
 this.setSize(new Dimension(400, 319));
 this.setFont(new java.awt.Font("SansSerif", 0, 11));
 this.setTitle("Copie d'une liste par sélection de chaque élément");
 this.setLayout(null);
 listInitial.setBackground(Color.yellow);
 listInitial.setBounds(new Rectangle(28, 41, 147, 171));
 listInitial.addMouseListener( new java.awt.event.MouseAdapter( )
 {
  public void mouseClicked(MouseEvent e)  {
   listInitial_mouseClicked(e);
  }
 }  );
 listCopie.setBackground(Color.cyan);
 listCopie.setBounds(new Rectangle(229, 43, 141, 166));
 listCopie.addItemListener( new java.awt.event.ItemListener( )   {
  public void itemStateChanged(ItemEvent e)  {
   listCopie_itemStateChanged(e);
  }
 }  );
 textFieldSaisie.setBackground(Color.yellow);
 textFieldSaisie.setText("");
 textFieldSaisie.setBounds(new Rectangle(31, 232, 145, 23));
 buttonRanger.setForeground(Color.black);
 buttonRanger.setLabel("Ranger dans la liste");
 buttonRanger.setBounds(new Rectangle(31, 259, 147, 23));
 buttonRanger.addActionListener( new java.awt.event.ActionListener( )
 {
  public void actionPerformed(ActionEvent e)  {
   buttonRanger_actionPerformed(e);
  }
 }  );
 textFieldVisu.setBackground(Color.cyan);
 textFieldVisu.setEditable(false);
 textFieldVisu.setText("");
 textFieldVisu.setBounds(new Rectangle(231, 230, 141, 27));
 label1.setText("Entrez un nom :");
```

Classe anonyme

Classe anonyme

Classe anonyme

```
        label1.setBounds( new Rectangle(33, 220, 131, 13));
        label2.setBounds( new Rectangle(42, 28, 109, 13));
        label2.setText("Liste de noms entrés");
        label3.setText("Liste de noms sélectionnés");
        label3.setBounds(new Rectangle(230, 30, 139, 13));
        this.add (textFieldVisu);
        this.add (buttonRanger);
        this.add (label1);
        this.add (label2);
        this.add (label3);
        this.add (listInitial);
        this.add (listCopie);
        this.add (textFieldSaisie);
      }

    protected void processWindowEvent(WindowEvent e){
      super.processWindowEvent(e);
      if (e.getID( ) == WindowEvent.WINDOW_CLOSING)
        System.exit(100);
    }

    void buttonRanger_actionPerformed(ActionEvent e) {
      if (textFieldSaisie.getText( ).length( ) != 0)
        listInitial.add (textFieldSaisie.getText( ));
    }

    void listInitial_mouseClicked(MouseEvent e) {
      if (e.getClickCount( ) ==1 & e.getButton( ) == MouseEvent.BUTTON1)
        listCopie.add (listInitial.getSelectedItem( ));
      else
        if(e.getClickCount( ) ==2 & listInitial.getSelectedIndex( ) !=-1)
          listInitial.remove(listInitial.getSelectedIndex( ));
    }

    void listCopie_itemStateChanged(ItemEvent e){
      if (e.getStateChange( ) == ItemEvent.SELECTED)
        textFieldVisu.setText(listCopie.getSelectedItem( ));
    }
}
```

Fermeture de la fenetre sur l'événement : WINDOW_CLOSING

Click sur le **buttonRanger** intercepté par événement de haut nibveau (sémantique) : **actionPerformed**.

Click et double-click dans **listInitial** interceptés par événement bas niveau : **mouseClicked**.

Click dans **listCopie** simulé par l'interception du changement d'item sélectionné (obligatoirement par un click gauche)

IHM - avec Swing

Java2

Composants lourds, composants légers

Selon les bibliothèques de composants visuels utilisées, AWT ou Swing, Java n'adopte pas la même démarche d'implantation. Ceci est dû à l'évidence une évolution rapide du langage qui contient des couches successives de concepts.

Les composants lourds

En java, comme nous l'avons vu au chapitre AWT, les composants dérivent tous de la classe java.awt.Component. Les composants awt sont liés à la plate-forme locale d'exécution, car ils sont implémentés en code natif du système d'exploitation hôte et la Java Machine y fait appel lors de l'interprétation du programme Java. Ceci signifie que dès lors que vous développez une interface AWT sous windows, lorsque par exemple cette interface s'exécute sous MacOs, l'**apparence visuelle** et le **positionnement** des différents composants (boutons,...) **changent**. En effet la fonction système qui dessine un bouton sous Windows ne dessine pas le même bouton sous MacOs et des chevauchements de composants peuvent apparaître si vous les placez au pixel près (***d'où le gestionnaire LayOutManager pour positionner les composants !***).

De tels composants dépendant du système hôte sont appelés en Java des composants lourds. En Java le composant lourd est identique en tant qu'objet Java et il est associé localement lors de l'exécution sur la plateforme hôte à un élément local dépendant du système hôte dénommé **peer**.

Tous les composants du package AWT sont des composants lourds.

Les composants légers

Par opposition aux composants lourds utilisant des **peer** de la machine hôte, les composants légers sont entièrement écrits en Java. En outre un tel composant léger n'est pas dessiné visuellement par le système, mais par Java. Ceci apporte une amélioration de portabilité et permet même de changer l'apparence de l'interface sur la même machine grâce au "look and feel". La classe lookAndFeel permet de déterminer le style d'aspect employé par l'interface utilisateur.

Les composants **Swing** (nom du package : **javax.swing**) sont pour la majorité d'entre eux des composants **légers**.

En Java on ne peut pas se passer de composants lourds (communiquant avec le système) car la Java Machine doit communiquer avec son système hôte. Par exemple la fenêtre étant l'objet

visuel de base dans les systèmes modernes elle est donc essentiellement liée au système d'exploitation et donc ce sera en Java un composant lourd.

Swing contient un minimum de composants lourds

Dans le package Swing le nombre de composants lourds est réduit au strict minimum soient 4 genres de fenêtres.

Les fenêtres Swing sont des composants lourds

Les fenêtres en Java Swing :

- **JFrame** à rapprocher de la classe **Frame** dans AWT
- **JDialog** à rapprocher de la classe **Dialog** dans AWT
- **JWindow** à rapprocher de la classe **Window** dans AWT
- **JApplet** à rapprocher de la classe **Applet** dans AWT

Hiérarchie de classe de ces composants de fenêtres :

```
java.lang.Object
 |
 +--java.awt.Component
      |
      +--java.awt.Container
           |
           +--java.awt.Window
           |     |
           |     +--javax.swing.JWindow
           |     |
           |     +--java.awt.Frame
           |     |     |
           |     |     +--javax.swing.JFrame
           |     +--java.awt.Dialog
           |           |
           |           +--javax.swing.JDialog
           +--java.awt.Panel
                 |
                 +--java.applet.Applet
                       |
                       +--javax.swing.JApplet
```

Le principe appliqué étant que si la fenêtre a besoin de communiquer avec le système, les composants déposés sur la fenêtre eux n'en ont pas la nécessité. C'est pourquoi tous les autres composants de **javax.swing** sont des composants légers. Pour utiliser les Swing, il suffit d'importer le package :

```
import javax.swing.*
```

Il est bien sûr possible d'utiliser des composants AWT et Swing dans la même application. Les événements sont gérés pour les deux packages par les méthodes de l'interface Listener du package **java.awt.event**.

Ce qui signifie que tout ce qui a été dit au chapitre sur les événements pour les composants **AWT** (***modèle de délégation du traitement de l'événement à un écouteur***) est intégralement reportable aux **Swing** sans aucune modification.

En général, les classes des composants **swing** étendent les fonctionnalités des classes des composants **AWT** dont elles héritent (plus de propriétés, plus d'événements,...).

Les autres composants Swing sont légers

Les composants légers héritent tous directement ou indirectement de la classe **javax.swing.JComponen**t :

```
java.lang.Object
 |
 +--java.awt.Component
     |
     +--java.awt.Container
         |
         +--javax.swing.JComponent
```

Dans un programme Java chaque composant graphique Swing (léger) doit donc disposer d'un conteneur de plus haut niveau sur lequel il doit être placé.

Afin d'assurer la communication entre les composants placés dans une fenêtre et le système, le package Swing organise d'une manière un peu plus complexe les relations entre la fenêtre propriétaires et ses composants.

Le JDK1.4.2 donne la liste suivante des composants légers héritant de JComponent : AbstractButton, BasicInternalFrameTitlePane, JColorChooser, JComboBox, JFileChooser, JInternalFrame, JInternalFrame.JDesktopIcon, JLabel, JLayeredPane, JList, JMenuBar, JOptionPane, JPanel, JPopupMenu, JProgressBar, JRootPane, JScrollBar, JScrollPane, JSeparator, JSlider, JSplitPane, JTabbedPane, JTable, JTableHeader, JTextComponent, JToolBar, JToolTip, JTree, JViewport.

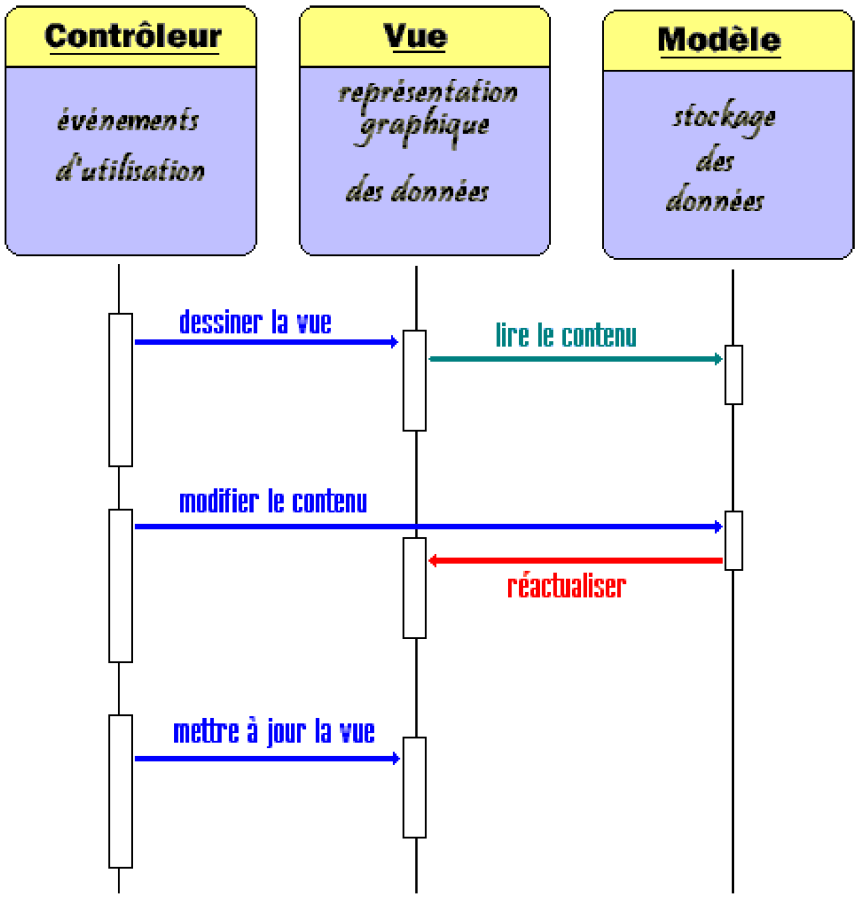
L'architecture Modèle-Vue-Contrôleur en général (MVC)

En technologie de conception orientée objet il est conseillé de ne pas confier trop d'actions à un seul objet, mais plutôt de répartir les différentes responsabilités d'actions entre plusieurs objets. Par exemple pour un composant visuel (bouton, liste etc...) vous déléguez la **gestion du style du composant à une classe** (ce qui permettra de changer facilement le style du composant sans intervenir sur le composant lui-même), vous stockez les données contenues dans le composant dans **une autre classe chargée de la gestion des données de contenu** (ce qui permet d'avoir une gestion décentralisée des données) .

Si l'on recense les caractéristiques communes aux composants visuels servant aux IHM (interfaces utilisateurs), on retrouve 3 constantes générales pour un composant :

- son contenu (les données internes, les données stockées, etc...)
- son apparence (style, couleur, taille, etc...)
- son comportement (essentiellement en réaction à des événements)

Diagramme de séquence UML des interactions MVC



Le schéma précédent représente l'architecture **Modèle-Vue-Contrôleur** (ou design pattern observateur-observé) qui réalise cette conception décentralisée à l'aide de 3 classes associées à chaque composant :

- Le **modèle** qui stocke le contenu, qui contient des méthodes permettant de modifier le contenu et qui n'est pas visuel.
- La **vue** qui affiche le contenu, est chargée de dessiner sur l'écran la forme que prendront les données stockées dans le modèle.
- Le **contrôleur** qui gére les interactions avec l'utilisateur .

Le pluggable look and feel

C'est grâce à cette architecture MVC, que l'on peut implémenter la notion de "**pluggable look and feel (ou plaf)**" qui entend séparer le modèle sous-jacent de la représentation visuelle de l'interface utilisateur. Le code Swing peut donc être réutilisé avec le même modèle mais changer de style d'interface dynamiquement pendant l'exécution.

Voici à titre d'exemple la même interface Java écrite avec des Swing et trois aspects différents (motif, métal, windows) obtenus pendant l'exécution en changeant son look and feel par utilisation de la classe UIManager servant à gérer le look and feel.

avec le système Windows sont livrés 3 look and feel standard : windows (apparence habituelle de windows), motif (apparence graphique Unix) et metal (apparence genre métallique).

Trois exemples de look and feel

Voici ci-après trois aspects de **la même interface utilisateur**, chaque aspect est changé durant l'exécution uniquement par l'appel des lignes de code associées (this représente la fenêtre JFrame de l'IHM) :

Lignes de code pour passer en IHM motif :

```
String UnLook = "com.sun.java.swing.plaf.motif.MotifLookAndFeel" ;
try  {
     UIManager.setLookAndFeel(UnLook); // assigne le look and feel choisi ici motif
     SwingUtilities.updateComponentTreeUI(this.getContentPane( ));  // réactualise le graphisme de l'IHM
     }
catch (Exception exc) {
      exc.printStackTrace( );
}
```

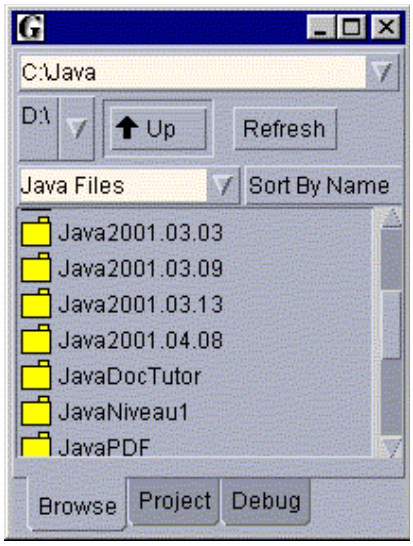
Lignes de code pour passer en IHM métal :

```
String UnLook = "javax.swing.plaf.metal.MetalLookAndFeel" ;
try  {
     UIManager.setLookAndFeel(UnLook); // assigne le look and feel choisi ici metal
     SwingUtilities.updateComponentTreeUI(this.getContentPane( ));  // réactualise le graphisme de l'IHM
     }
catch (Exception exc) {
      exc.printStackTrace( );
}
```

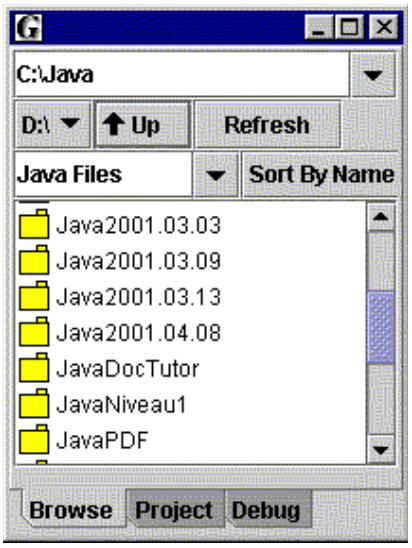
Lignes de code pour passer en IHM Windows :

```
String UnLook = "com.sun.java.swing.plaf.windows.WindowsLookAndFeel" ;
try  {
     UIManager.setLookAndFeel(UnLook); // assigne le look and feel choisi ici windows
     SwingUtilities.updateComponentTreeUI(this.getContentPane( ));  // réactualise le graphisme de l'IHM
     }
catch (Exception exc) {
      exc.printStackTrace( );
}
```

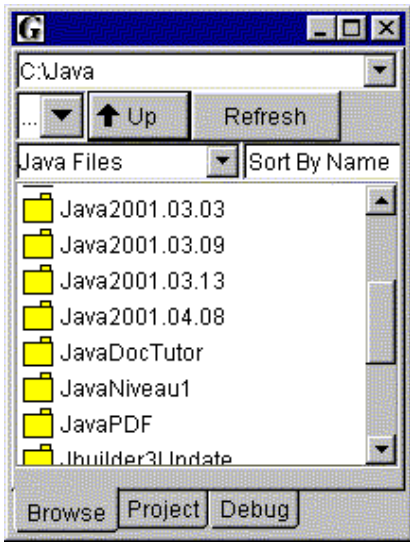
Aspect motif de l'IHM



Aspect métal de l'IHM



Aspect Windows de l'IHM :



Les swing reposent sur MVC

Les composants de la bibliothèque **Swing** adoptent tous cette architecture de type **MVC (Modèle-Vue-Contrôleur)** qui sépare le stockage des données, leur représentation et les interactions possibles avec les données, les composants sont associés à différentes interfaces de modèles de base. Toutefois les swing pour des raisons de souplesse **ne respectent pas strictement l'architecture MVC** que nous venons de citer :

- Le **Modèle**, chargé de stocker les données, qui permet à la vue de lire son contenu et informe la vue d'éventuelles modifications est bien ***représenté par une classe***.
- La **Vue**, permettant une représentation des données (nous pouvons l'assimiler ici à la représentation graphique du composant) peut être ***répartie sur plusieurs classes***.
- Le **Contrôleur**, chargé de gérer les interactions de l'utilisateur et de propager des modifications vers la vue et le modèle peut aussi être réparti sur plusieurs classes, voir même dans des ***classes communes à la vue et au contrôleur***.

Exemple de quelques interfaces de modèles rencontrées dans la bibliothèque Swing

Identificateur de la classe de modèle	*Utilisation*
ListModel	Modèle pour les listes (JList ...)
ButtonModel	Modèle d'état pour les boutons (JButton...)
Document	Modèle de document (JTextField...

La mise en oeuvre des composants Swing ne requiert pas systématiquement l'utilisation des modèles. Il est ainsi généralement possible d'initialiser un composant Swing à l'aide des données qu'il doit représenter. Dans ce cas , le composant exploite un modèle interne par défaut pour stocker les données.

Le composant javax.swing.Jlist

Dans le cas du JList le recours au modèle est impératif, en particulier une vue utilisant le modèle ListModel pour un jList enregistrera un écouteur sur l'implémentation du modèle et effectuera des appels à **getSize**( ) et **getElementAt**( ) pour obtenir le nombre d'éléments à représenter et les valeurs de ces éléments.

Dans l'exemple suivant, l'un des constructeurs de JList est employé afin de définir l'ensemble des données à afficher dans la liste, on suppose qu'il est associé à un modèle dérivant de **ListModel** déjà défini auparavant. L'appel à **getModel**( ) permet d'obtenir une référence sur l'interface ListModel du modèle interne du composant :

```
JList  Jlist1 = new JList(new Objet[] {"un","deux","trois"});
ListModel  modeldeliste = jList1.getModel( );
System.out.println ("Élément       0  :   "  +  modeldeliste.getElementAt(0));
System.out.println ("Nb éléments : "+ modeldeliste.getSize( ))
```

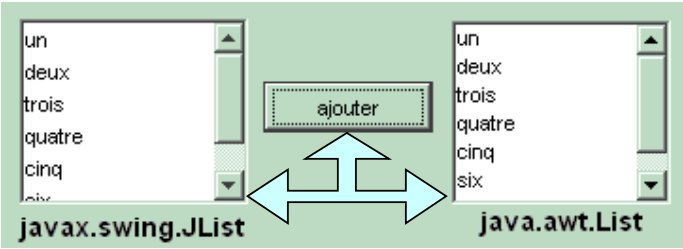
Pour mettre en oeuvre les modèles et les fournir aux composants on utilise la méthode

setModel( ) (***public void*** *setModel (ListModel model) { }*) du composant. Comme *ListModel* est une interface, il nous faut donc implémenter cette interface afin de passer un paramètre effectif (*ListModel model*), nous choisissons la classe *DefaultListModel* qui est une implémentation de l'interface *ListModel* par le biais d'un vecteur. Il est ainsi possible d'instancier le *ListModel* d'agir sur le modèle (ajout, suppression d'éléments) et de l'enregistrer auprès du composant adéquat grâce à **setModel**( ).

Le listing suivant illustre la mise en oeuvre de *DefaultListModel( )* pour un JList :

JList jList1 = **new** JList( ); DefaultListModel dlm = **new** DefaultListModel ( );	***// instanciations d'un JList et d'un modèle.***
dlm.addElement ("un"); dlm.addElement ("deux"); dlm.addElement ("trois"); dlm.addElement ("quatre");	***// actions d'ajout d'éléments dans le modèle.***
jList1.setModel(dlm);	***// enregistrement du modèle pour le JList.***
dlm.removeElementAt(1); dlm.removeRange(0,2); dlm.add(0,"Toto");	***// actions de suppression et d'ajout d'éléments dans le modèle.***

Comparaison awt.List et swing.JList

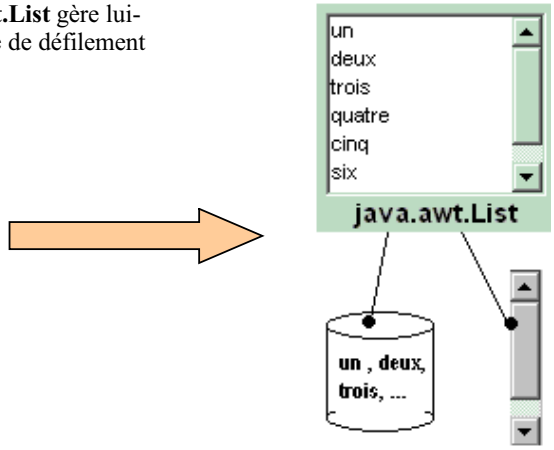


Une IHM dans laquelle, le bouton Ajouter, insère 7 chaînes dans chacun des deux composants.

L'apparence est la même lors de l'affichage des données dans chacun des composants, dans le code il y a une totale différence de gestion entre le composant List et le composant Jlist.

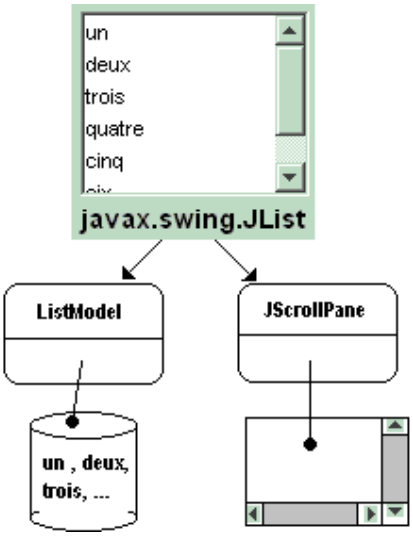
Comme en Delphi le composant **java.awt.List** gère lui-même le stockage des données, et la barre de défilement verticale.

```
List list1 = new List();
list1.add("un");
list1.add("deux");
list1.add("trois");
list1.add("quatre");
list1.add("cinq");
list1.add("six");
list1.add("sept");
```



Le composant **javax.swing.Jlist** délègue le stockage des données à un **modèle** et la gestion de la barre de défilement verticale à un autre composant dédié : un **javax.swing.JscrollPane**.

```
JList jList1 = new JList();
DefaultListModel dlm = new DefaultListModel();
JScrollPane jScrollPane1 = new JScrollPane();
jList1.setModel(dlm);
jScrollPane1.getViewport().add(jList1);
dlm.addElement("un");
dlm.addElement("deux");
dlm.addElement("trois");
dlm.addElement("quatre");
dlm.addElement("cinq");
dlm.addElement("six");
dlm.addElement("sept");
```



Le composant javax.swing. JTextPane

Soit le code suivant :

```
Style styleTemporaire;
StyleContext leStyle = new StyleContext();
Style parDefaut = leStyle.getStyle(StyleContext.DEFAULT_STYLE);
Style styleDuTexte = leStyle.addStyle("DuTexte1", parDefaut);
StyleConstants.setFontFamily(styleDuTexte, "Courier New");
StyleConstants.setFontSize(styleDuTexte, 18);
StyleConstants.setForeground(styleDuTexte, Color.red);
styleTemporaire = leStyle.addStyle("DuTexte2", styleDuTexte);
StyleConstants.setFontFamily(styleTemporaire, "Times New Roman");
StyleConstants.setFontSize(styleTemporaire, 10);
StyleConstants.setForeground(styleTemporaire, Color.blue);
styleTemporaire = leStyle.addStyle("DuTexte3", styleDuTexte);
StyleConstants.setFontFamily(styleTemporaire, "Arial Narrow");
StyleConstants.setFontSize(styleTemporaire, 14);
StyleConstants.setBold(styleTemporaire, true);
StyleConstants.setForeground(styleTemporaire, Color.magenta);
DefaultStyledDocument  format = new DefaultStyledDocument(leStyle);
```

Caractérisation du style n°1 du document

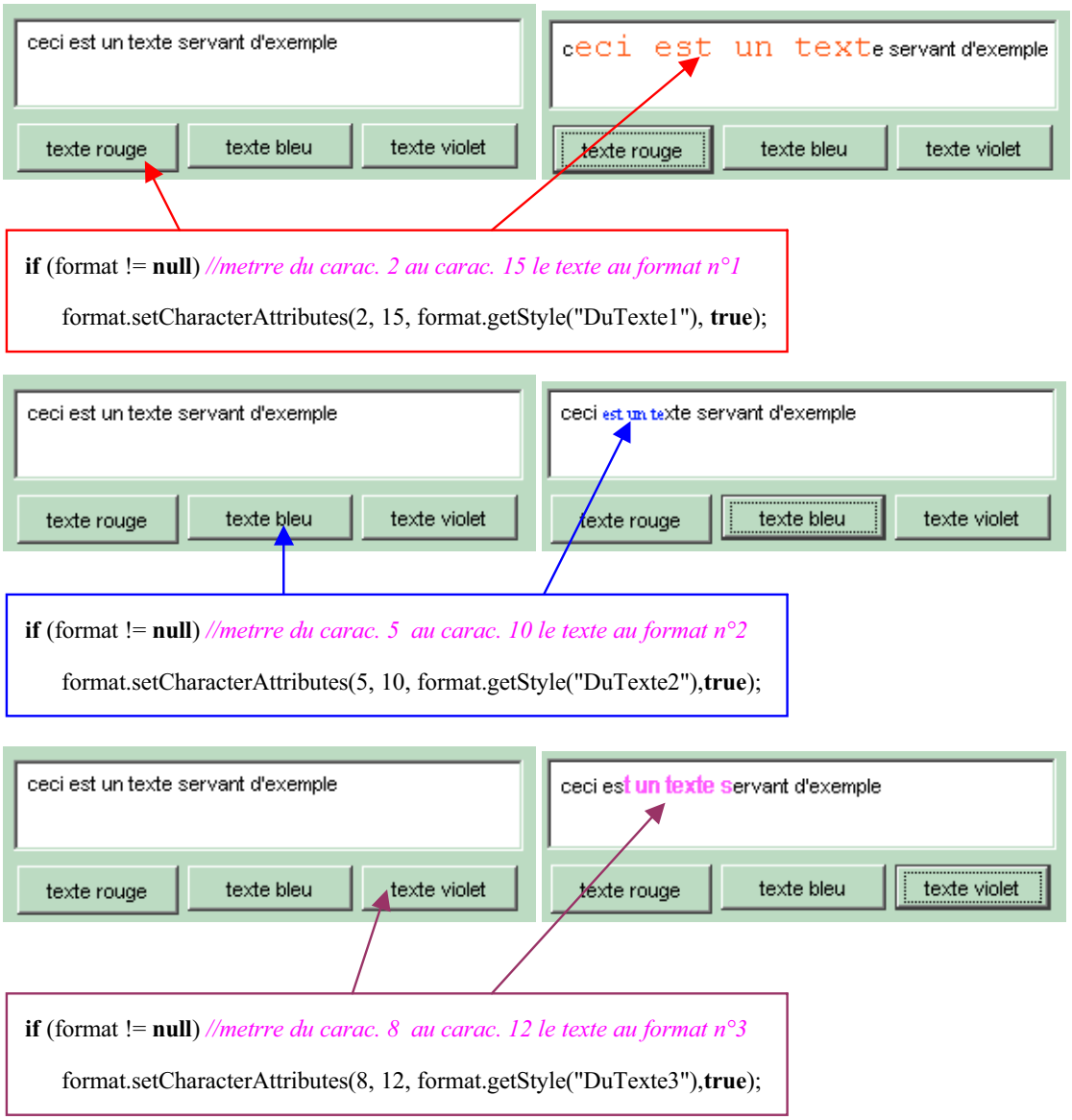
Caractérisation du style n°2 du document

Caractérisation du style n°3 du document

Le document gère les styles du JtextPane.

```
jTextPane1.setDocument(format);
```

Un composant de classe **JTextPane** est chargé d'afficher du texte avec plusieurs styles d'attributs (police, taille, couleur,…), la gestion proprement dite des styles est déléguée à un objet de classe DefaultStyledDocument que nous avons appelé **format** (gestion MVC) :

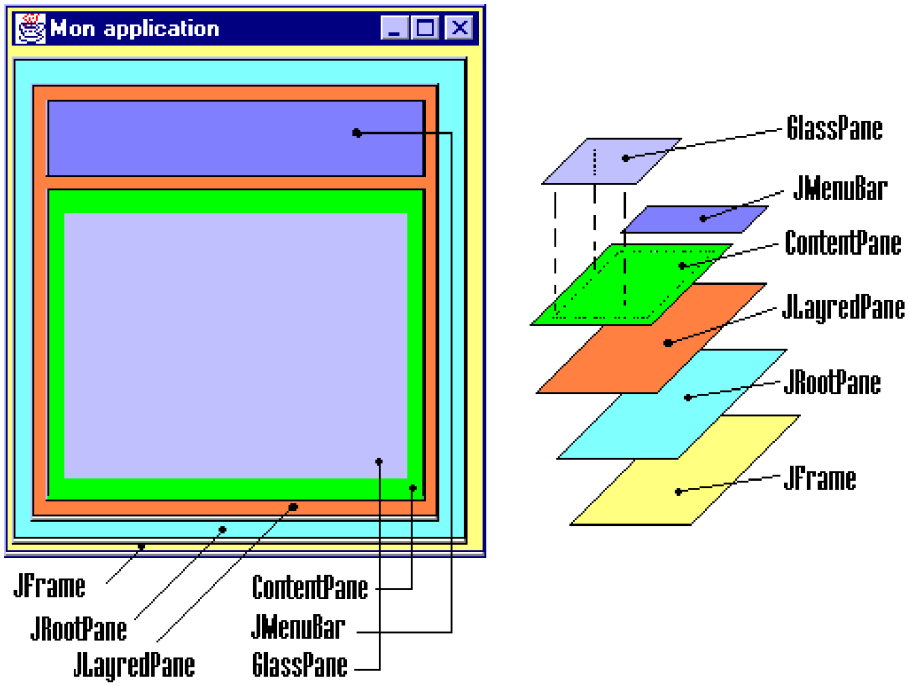


Chaque bouton lance l'application d'un des trois styles d'attributs à une partie du texte selon le modèle de code suivant :

- SetCharacterAttributes (<n° cardébut>,<n° carfin>,< le style>,**true**);
- ensuite automatiquement, le **JTextPane** informé par l'objet format qui gère son modèle de style de document, affiche dans l'image de droite le changement du style .

En Java le JFrame est un conteneur de composants (barre de menus, boutons etc...) qui dispose de 4 niveaux de superposition d'objets à qui est déléguée la gestion du contenu du JFrame.

Système de conteneur pour afficher dans une Fenêtre ou une Applet



Notons que le **JRootPane**, le **JLayredPane** et le **GlassPane** sont utilisés par Swing pour implémenter le look and feel, ils n'ont donc pas à être considérés dans un premier temps par le développeur, la couche qui nous intéresse afin de déposer un composant sur une fenêtre JFrame est la couche **ContentPane** instanciation de la classe Container. Les rectangles colorés imbriqués ci-haut, sont dessinés uniquement à titre pédagogique afin d'illustrer l'architecture en couche, ils ne représentent pas des objets visuels dont les tailles seraient imbriquées. En effet le GlassPane bien que dessiné plus petit (pour mieux le situer) prend par exemple toute la taille du Contentpane.

Swing instancie **automatiquement** tous ces éléments dès que vous instanciez un JFrame (à part JMenuBar qui est facultatif et qu'il faut instancier manuellement).

Pour ajouter des composants à un JFrame, il faut les ajouter à son objet ContentPane (la référence de l'objet est obtenu par la méthode getContentPane( ) du JFrame).

Exemple d'ajout d'un bouton à une fenêtre

Soit à ajouter un bouton de la classe des JButton sur une fenêtre de la classe des JFrame :

```
JFrame LaFenetre = new JFrame( ) ; // instanciation d'un JFrame
JButton UnBouton = new JButton( ) ; // instanciation d'un JButton
Container ContentPane = LaFenetre.getContentPane( ) ; // obtention de la référence du contentPane du JFrame
....
ContentPane.setLayout(new XYLayout( )); // on choisi le layout manager du ContentPane
ContentPane.add(UnBouton) ; // on dépose le JButton sur le JFrame à travers son ContentPane
....
```

Attention : avec AWT le dépôt du composant s'effectue directement sur le conteneur. Il faut en outre éviter de mélanger des AWT et des Swing sur le même conteneur.

AWT	Swing
Frame LaFenetre = **new** Frame( ) ; **Button** UnBouton = **new** Button( ) ; LaFenetre.add(UnBouton) ;	**JFrame** LaFenetre = **new** JFrame( ) ; **JButton** UnBouton = **new** JButton( ) ; **Container** ContentPane = LaFenetre.**getContentPane**( ) ; ContentPane.add(UnBouton) ;

Conseils au débutant

L'IDE Java JGrasp de l'université d'Auburn (téléchargement gratuit à Auburn) permet le développement d'applications pédagogiques dès que vous avez installé la dernière version du JDK (téléchargement gratuit chez Sun). Si vous voulez bénéficier de la puissance équivalente à Delphi pour écrire des applications fenêtrées il est conseillé d'utiliser un RAD (sauf à préférer les réexécutions fastidieuses pour visualiser l'état de votre interface).

JBuilder est un outil RAD particulièrement puissant et convivial qui aide au développement d'application Java avec des IHM (comme Delphi le fait avec pascal). La société Borland qui s'est spécialisée dans la création de plate-formes de développement est dans le peloton de tête avec ce RAD sur le marché des IDE (une version personnelle est en téléchargement gratuite chez Borland).

Nous recommandons donc au débutant en Java d'adopter ces deux outils dans cet ordre.

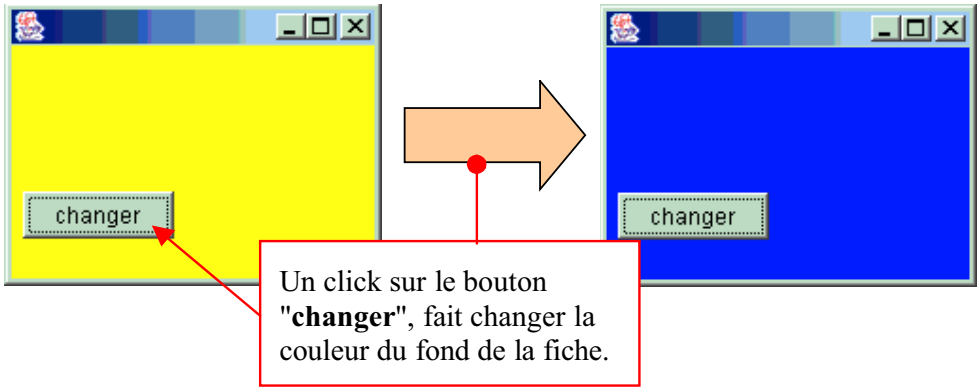
Dans les pages qui suivent nous reprenons les exercices écrits avec les Awt en utilisant leur correspondant Swing. Comme nous avons déjà expliqué les sources nous ne donnerons des indications que lorsque l'utilisation du composant Swing induit des lignes de code différentes.

Exercices IHM - JFrame de Swing

Soit l'IHM suivante composée d'une fiche **Fenetre** de classe **JFrame**, d'un bouton **jButton1** de classe **JButton**.

L'IHM réagit uniquement au click de souris :

- Le **jButton1** de classe **JButton** réagit au simple click et fait passer le fond de la fiche à la couleur bleu.
- La fiche de classe **JFrame** est sensible au click de souris pour sa fermeture et arrête l'application.




```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Fenetre extends JFrame {
 Container contentPane;
 JButton jButton1 = new JButton( );

 public Fenetre() {
  enableEvents (AWTEvent.WINDOW_EVENT_MASK);
  contentPane =  this.getContentPane( );
  jButton1.setBounds(new Rectangle(10, 80, 80, 25));
  jButton1.setText("changer");
  jButton1.addMouseListener(  new java.awt.event.MouseAdapter( )
                              {
                                 public void mouseClicked(MouseEvent e)  {
                                    GestionnaireClick(e);
                                 }
                              }
  );
  contentPane.setLayout(null);
  this.setSize(new Dimension(200, 150));
  this.setTitle("");
  contentPane.setBackground(Color.yellow);
  contentPane.add(jButton1, null);
 }
```

On récupère dans la variable contentPane, la référence sur le Container renvoyée par la méthode getContentPane.

Version avec une classe anonyme d'écouteur dérivant des MouseAdapter. (identique Awt)

```
protected void processWindowEvent(WindowEvent e) {
 super.processWindowEvent(e);
 if (e.getID( ) == WindowEvent.WINDOW_CLOSING)
  System.exit(100);
 }

 void GestionnaireClick(MouseEvent e) {
  this.contentPane.setBackground(Color.blue);
 }
}

public class ExoSwing {

   public static void main ( String [ ] x ) {
    Fenetre fen = new Fenetre ( );
   }
}
```

Fermeture de la JFrame des Swing identique à la Frame des Awt.

La bibliothèque Swing apporte une amélioration de confort dans l'écriture du code fermeture d'une fenêtre de classe JFrame en ajoutant dans la classe JFrame une nouvelle méthode :

public void setDefaultCloseOperation(**int** operation)

Cette méthode indique selon la valeur de son paramètre de type int, quelle opération doit être effectuée lors que l'on ferme la fenêtre.

Les paramètres possibles sont au nombre quatre et sont des champs static de classe :

WindowConstants.DO_NOTHING_ON_CLOSE
WindowConstants.HIDE_ON_CLOSE
WindowConstants.DISPOSE_ON_CLOSE

JFrame.EXIT_ON_CLOSE

Dans la classe : **WindowConstants**

Dans la classe : **JFrame**

C'est ce dernier que nous retenons pour faire arrêter l'exécution del'application lors de la fermeturede la fenêtre. Il suffit d'insérer dans le constructeur de fenêtre la ligne qui suit :

setDefaultCloseOperation(**JFrame**.EXIT_ON_CLOSE);

Reprise du code de Fenetre avec cette modification spécifique aux JFrame

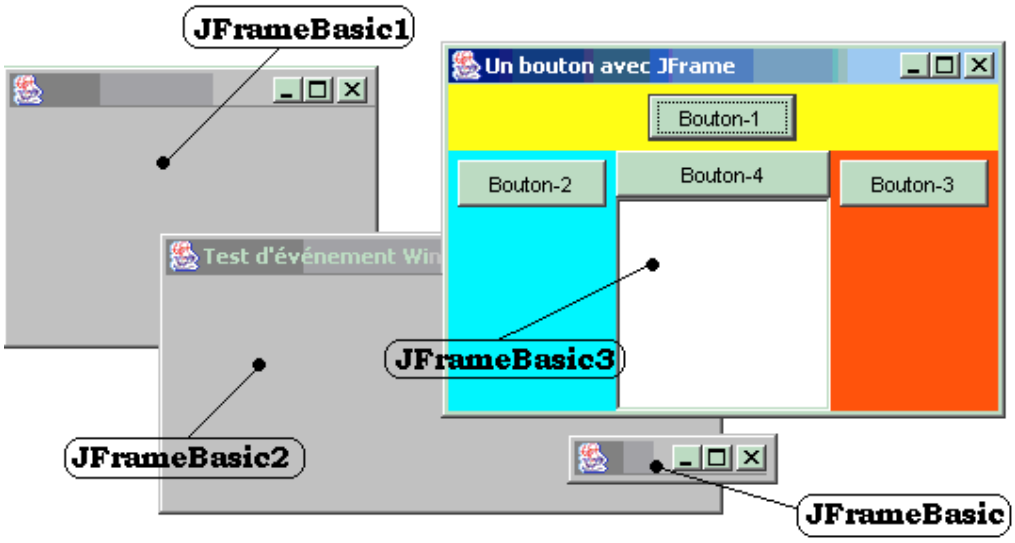
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Fenetre extends JFrame {
 Container contentPane;
 JButton jButton1 = new JButton( );

 public Fenetre( ) {
  contentPane = this.getContentPane( );
```

```
jButton1.setBounds(new Rectangle(10, 80, 80, 25));
jButton1.setText("changer");
jButton1.addMouseListener (   new java.awt.event.MouseAdapter( )
                              {
                                 public void mouseClicked(MouseEvent e)  {
                                    GestionnaireClick(e);
                                 }
                              }
);
contentPane.setLayout(null);
this.setSize(new Dimension(200, 150));
this.setTitle("");
contentPane.setBackground(Color.yellow);
contentPane.add(jButton1, null);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}

void GestionnaireClick(MouseEvent e)  {
  this.setBackground(Color.blue);
}
}
```

Fermeture de la fenêtre spécifique à la classe JFrame des Swing.

Soit l'IHM suivante composée de quatre fiches de classe **JFrame** nommées **JFrameBasic, JFrameBasic1, JFrameBasic2, JFrameBasic3**. Elle permet d'explorer le comportement d'événements de la classe WindowEvent sur une JFrame ainsi que la façon dont une JFrame utilise un layout

- Le **JFrameBasic2** de classe **JFrame** réagit à la fermeture, à l'activation et à la désactivation.
- Le **JframeBasic3** de classe **JFrame** ne fait que présenter visuellement le résultat d'un BorderLayout.



```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

 public class JFrameBasic extends JFrame  {
   JFrameBasic( )    {
     this.setVisible(true);
   }
 }

public class JFrameBasic1 extends JFrame {
  JFrameBasic1( )   {
    this.setVisible(true);
    this.setBounds(100,100,200,150);
  }
}


public class JFrameBasic2 extends JFrame {
  JFrameBasic2( )   {
    this.setVisible(true);
    this.setBounds(200,200,300,150);
    this.setTitle("Test d'événement Window");
    setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);
  }
  protected void processWindowEvent(WindowEvent e)   {
    super.processWindowEvent(e);
    if (e.getID( ) == WindowEvent.WINDOW_CLOSING)    {
      System.out.println("JFrameBasic2 / WINDOW_CLOSING = "+WindowEvent.WINDOW_CLOSING);
      dispose ( );
    }
    if (e.getID( ) == WindowEvent.WINDOW_CLOSED)    {
      System.out.print("JFrameBasic2 / WINDOW_CLOSED = "+WindowEvent.WINDOW_CLOSED);
      System.out.println(" => JFrameBasic2 a été détruite !");
    }
    if (e.getID( ) == WindowEvent.WINDOW_ACTIVATED)
     System.out.println("JFrameBasic2 / WINDOW_ACTIVATED = " + WindowEvent.WINDOW_ACTIVATED );
    if (e.getID( ) == WindowEvent.WINDOW_DEACTIVATED)
     System.out.println("JFrameBasic2 / WINDOW_DEACTIVATED = "+WindowEvent.WINDOW_DEACTIVATED);
  }

  public class JFrameBasic3 extends JFrame    {
    JButton Bout1 = new JButton("Bouton-1");
    JButton Bout2 = new JButton("Bouton-2");
    JButton Bout3 = new JButton("Bouton-3");
    JButton Bout4 = new JButton("Bouton-4");
    JTextArea jTextArea1 = new JTextArea();

    JFrameBasic3( )   {
      JPanel Panel1 = new JPanel( );
```

```
        JPanel Panel2 = new JPanel( );
        JPanel Panel3 = new JPanel( );
        JPanel Panel4 = new JPanel( );
        JScrollPane jScrollPane1 = new JScrollPane( );
        jScrollPane1.setBounds(20,50,50,40);
        this.setBounds(450,200,300,200);
        jScrollPane1.getViewport( ).add(jTextArea1);
        this.setTitle("Un bouton avec JFrame");
        Bout1.setBounds(5, 5, 60, 30);
        Panel1.add(Bout1);
        Bout2.setBounds(10, 10, 60, 30);
        Panel2.add(Bout2);
        Bout3.setBounds(10, 10, 60, 30);
        Panel3.add(Bout3);
        Bout4.setBounds(10, 10, 60, 30);
        Panel4.setLayout(new BorderLayout( ));
        Panel4.add(Bout4, BorderLayout.NORTH);
        Panel4.add(jScrollPane1, BorderLayout.CENTER);
        Panel1.setBackground(Color.yellow);
        Panel2.setBackground(Color.cyan);
        Panel3.setBackground(Color.red);
        Panel4.setBackground(Color.orange);
        this.getContentPane().setLayout(new BorderLayout( )); //specifique JFrame
        this.getContentPane().add(Panel1, BorderLayout.NORTH); //specifique JFrame
        this.getContentPane().add(Panel2, BorderLayout.WEST); //specifique JFrame
        this.getContentPane().add(Panel3, BorderLayout.EAST); //specifique JFrame
        this.getContentPane().add(Panel4, BorderLayout.CENTER); //specifique JFrame
        this.setVisible(true);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

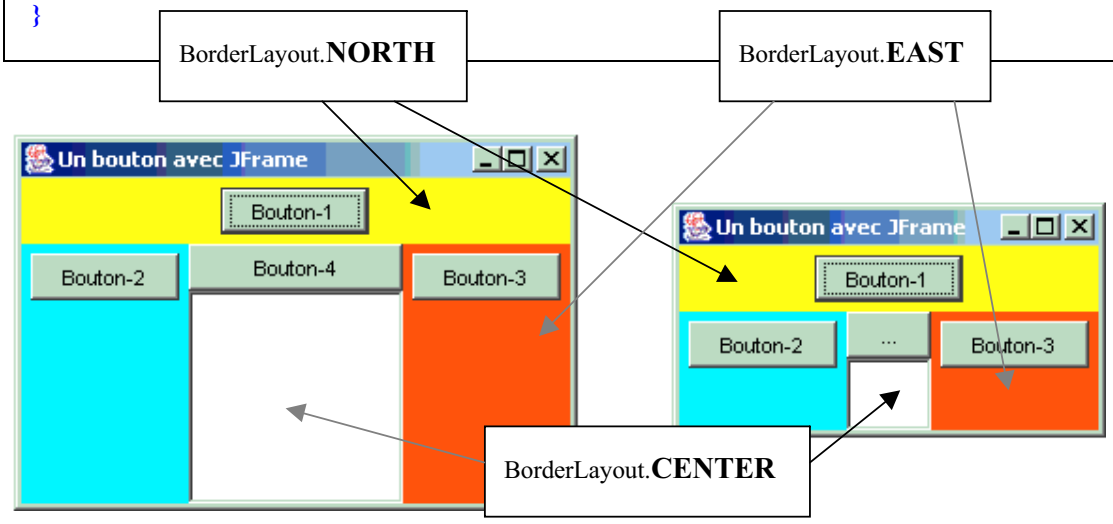
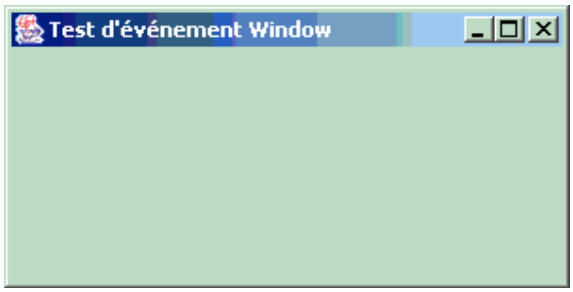


fig-repositionnement automatique des quatre Jpanel grâce au BorderLayout

Au démarrage voici les affichages consoles (la JFrameBasic2 est en arrière plan)

> *JFrameBasic2 / WINDOW_ACTIVATED = 205*
> *JFrameBasic2 / WINDOW_DEACTIVATED = 206*

En cliquant sur JFrameBasic2 elle passe au premier plan



voici l'affichages console obtenu :

> *JFrameBasic2 / WINDOW_ACTIVATED = 205*

En cliquant sur le bouton de fermeture de JFrameBasic2 elle se ferme mais les autres fenêtres restent, voici l'affichages console obtenu :

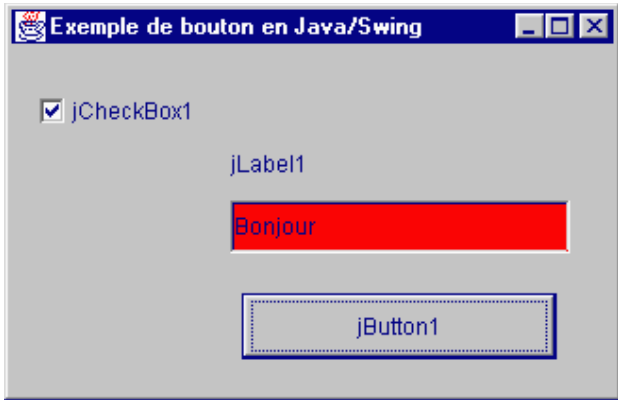
JFrameBasic2 / WINDOW_CLOSING = 201
JFrameBasic2 / WINDOW_DEACTIVATED = 206
JFrameBasic2 / WINDOW_CLOSED = 202 => JFrameBasic2 a été détruite !

Exemple - JButton de Swing

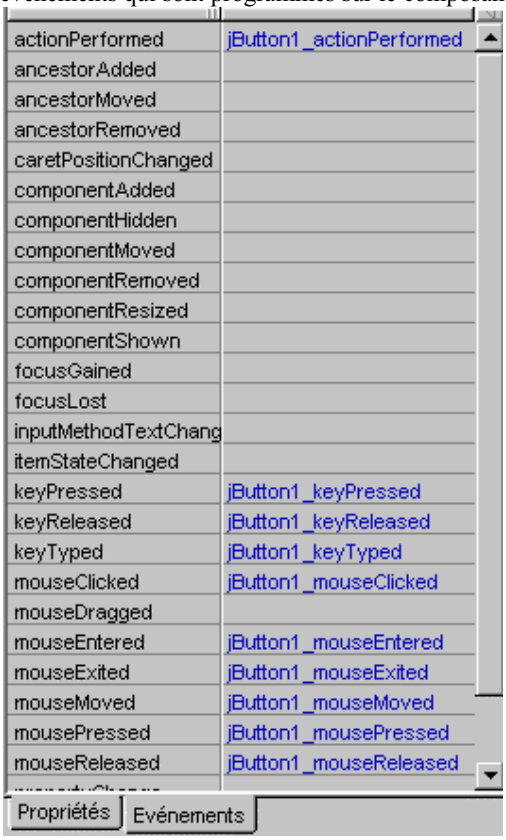
Code java généré par JBuilder

Objectif : Application simple Java utilisant les événements de souris et de clavier sur un objet de classe JButton.

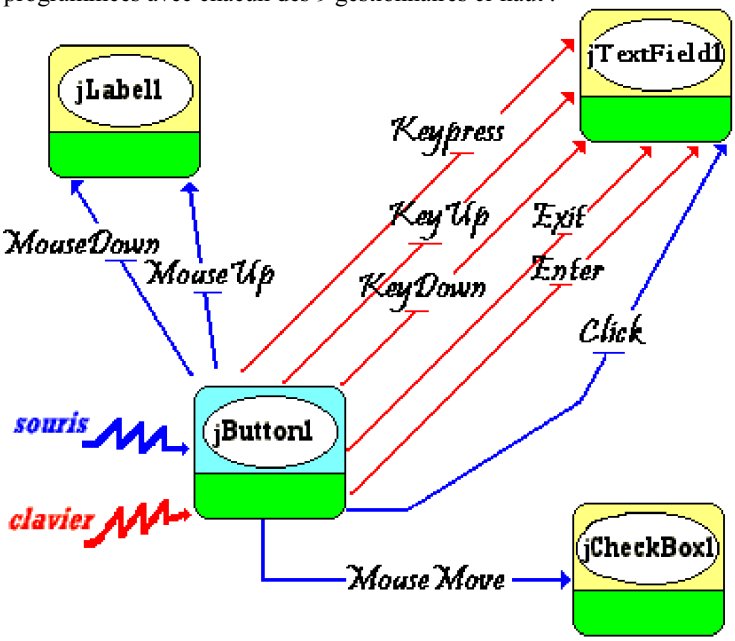
La fenêtre comporte un bouton (JButton jButton1), une étiquette (**JLabel** jLabel1), une case à cocher (**JCheckBox** jCheckBox1) et un éditeur de texte mono-ligne (**JTextField** jTextField1) :



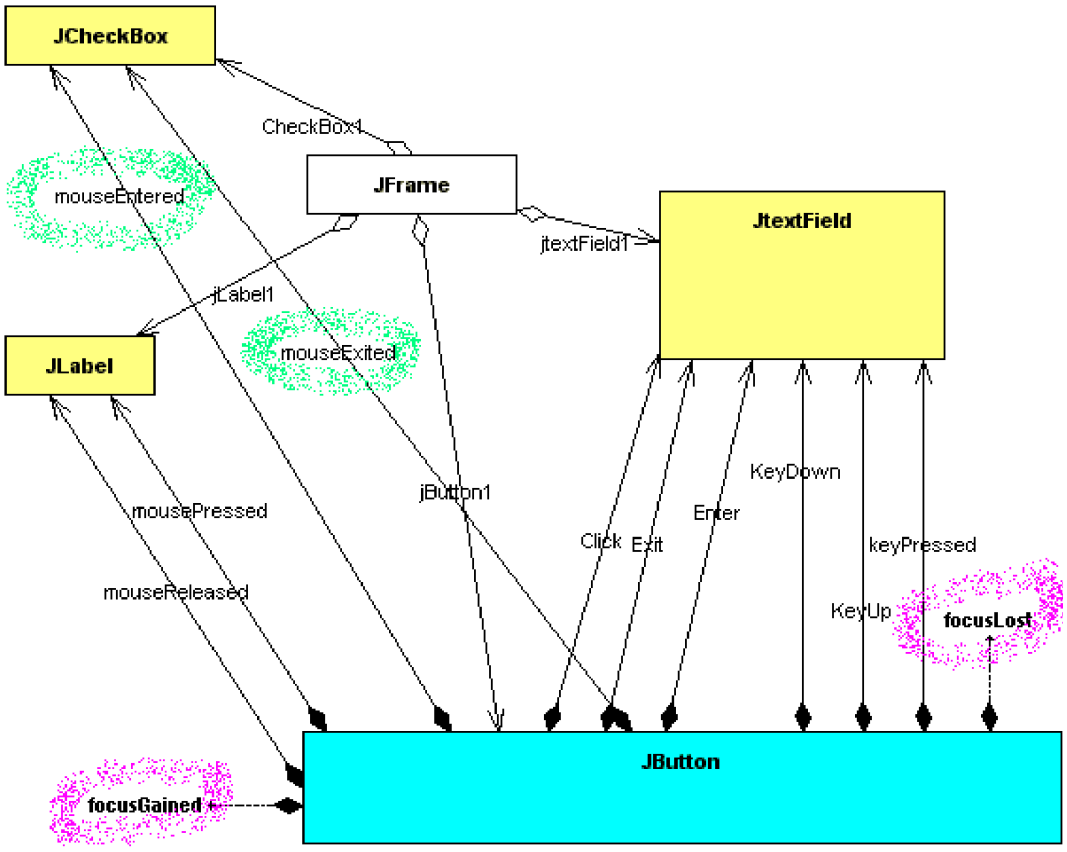
Voici les 10 gestionnaires d'événements qui sont programmés sur le composant jButton de classe **JButton**:



Voici le diagramme événementiel des actions de souris et de clavier sur le bouton jButton1. Ces 9 actions sont programmées avec chacun des 9 gestionnaires ci-haut :

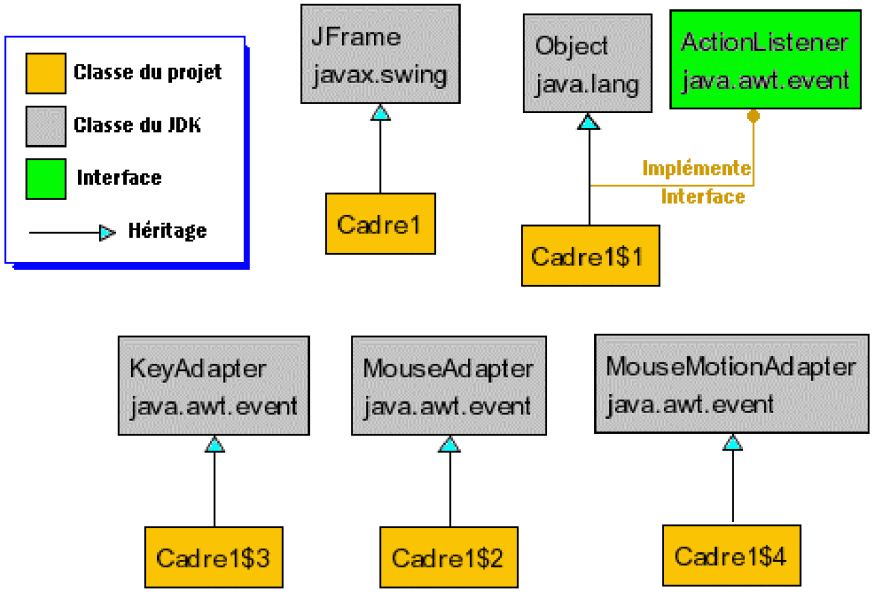


Les actions exit et enter sont représentées en Java par les événements focuGained et focusLost pour le clavier et par les événements mouseEntered et mouseExited pour la souris,. Il a été choisi de programmer les deux événements de souris dans le code ci-dessous.



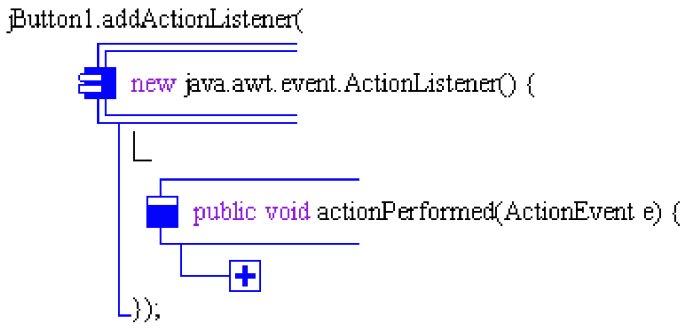
En Java

Comme en java tous les événements sont interceptés par des objets écouteurs, ci-dessous nous donnons les diagrammes UML des classes utilisées par le programme qui est proposé :

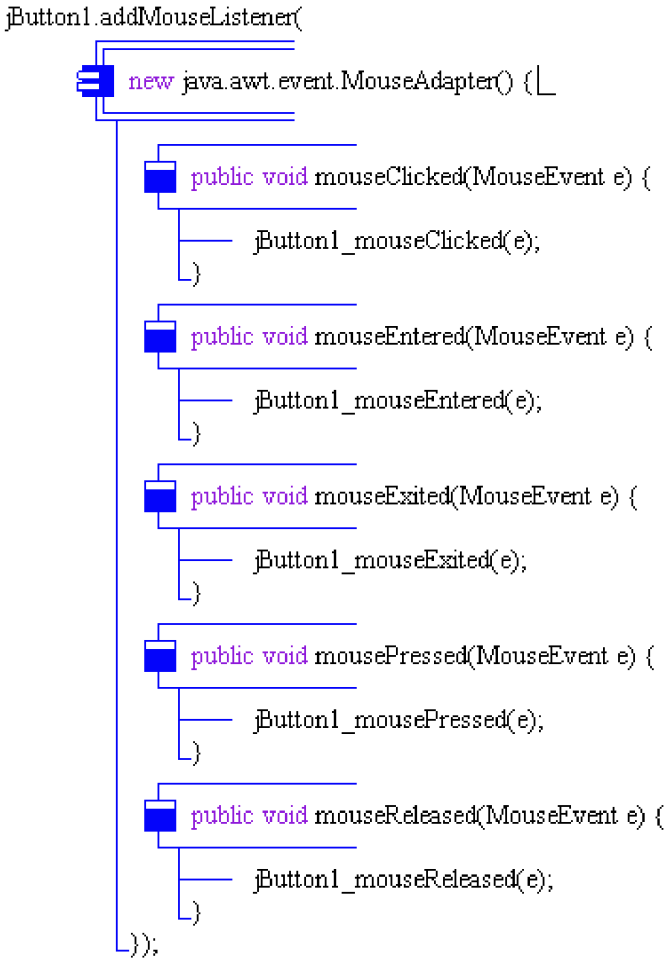


Rappelons que les classes Cadre1$1, Cadre1$2, ... sont la notation des classes anonymes créées lors de la déclaration de l'écouteur correspondant, Java 2 crée donc dynamiquement un objet écouteur interne (dont la référence n'est pas disponible). Ci-dessous les diagrammes jGrasp des quatre classes anonymes cadre1$1, Cadre1$2, Cadre1$3 et Cadre1$4 :

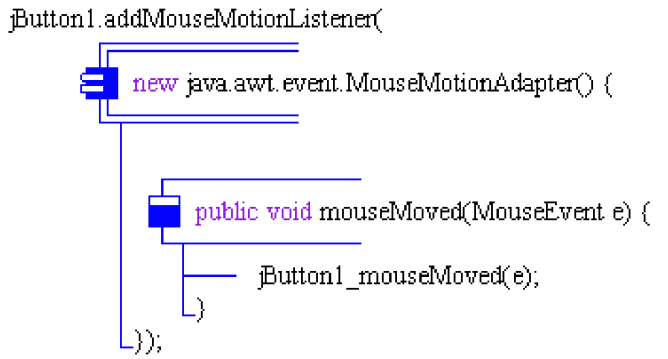
Cadre1$1:



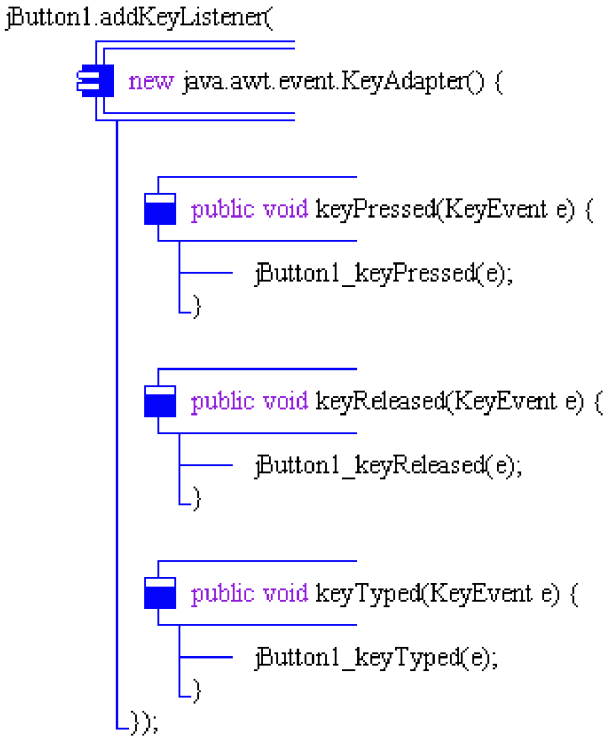
Cadre1$2:



Cadre1$4:



Cadre1$3:



Enfin pour terminer, voici le listing Java/Swing complet de la classe représentant la fenêtre :

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Cadre1 extends JFrame {
  JButton jButton1 = new JButton();
  JTextField jTextField1 = new JTextField();
  JLabel jLabel1 = new JLabel();
  JCheckBox jCheckBox1 = new JCheckBox();

  //Construire le cadre
  public Cadre1() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try  {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }

  //Initialiser le composant
  private void jbInit() throws Exception  {
    jButton1.setText("jButton1");
    jButton1.addActionListener(new java.awt.event.ActionListener() {

      public void actionPerformed(ActionEvent e) {
```

```
      jButton1_actionPerformed(e);
    }
  });
  jButton1.addMouseListener(new java.awt.event.MouseAdapter() {

    public void mouseClicked(MouseEvent e) {
      jButton1_mouseClicked(e);
    }

    public void mouseEntered(MouseEvent e) {
      jButton1_mouseEntered(e);
    }

    public void mouseExited(MouseEvent e) {
      jButton1_mouseExited(e);
    }

    public void mousePressed(MouseEvent e) {
      jButton1_mousePressed(e);
    }

    public void mouseReleased(MouseEvent e) {
      jButton1_mouseReleased(e);
    }
  });

  jButton1.addKeyListener(new java.awt.event.KeyAdapter() {

    public void keyPressed(KeyEvent e) {
      jButton1_keyPressed(e);
    }

    public void keyReleased(KeyEvent e) {
      jButton1_keyReleased(e);
    }

    public void keyTyped(KeyEvent e) {
      jButton1_keyTyped(e);
    }
  });

  jButton1.addMouseMotionListener(new java.awt.event.MouseMotionAdapter() {

    public void mouseMoved(MouseEvent e) {
      jButton1_mouseMoved(e);
    }
  });
  this.getContentPane().setLayout(null);
  this.setSize(new Dimension(327, 211));
  this.setTitle("Exemple de bouton en Java/Swing");
  jTextField1.setText("jTextField1");
  jTextField1.setBounds(new Rectangle(116, 82, 180, 28));
  jLabel1.setText("jLabel1");
  jLabel1.setBounds(new Rectangle(116, 49, 196, 26));
  jCheckBox1.setText("jCheckBox1");
  jCheckBox1.setBounds(new Rectangle(15, 22, 90, 25));
  this.getContentPane().add(jTextField1, null);
  this.getContentPane().add(jButton1, null);
  this.getContentPane().add(jCheckBox1, null);
  this.getContentPane().add(jLabel1, null);
}
```

```
//Remplacé (surchargé) pour pouvoir quitter lors de System Close
protected void processWindowEvent(WindowEvent e) {
  super.processWindowEvent(e);
  if(e.getID() == WindowEvent.WINDOW_CLOSING) {
    System.exit(0);
  }
}

void jButton1_mouseMoved(MouseEvent e) {
  jCheckBox1.setSelected(true);
}

void jButton1_keyPressed(KeyEvent e) {
 jTextField1.setText("Bonjour");
}

void jButton1_keyReleased(KeyEvent e) {
 jTextField1.setText("salut");
}

void jButton1_keyTyped(KeyEvent e) {
 jTextField1.setForeground(Color.blue);
}

void jButton1_mouseClicked(MouseEvent e) {
 jLabel1.setText("Editeur de texte");
}

void jButton1_mouseEntered(MouseEvent e) {
   jTextField1.setBackground(Color.red);
}

void jButton1_mouseExited(MouseEvent e) {
  jTextField1.setBackground(Color.green);
}

void jButton1_mousePressed(MouseEvent e) {
 jLabel1.setText("La souris est enfoncée");
}

void jButton1_mouseReleased(MouseEvent e) {
 jLabel1.setText("La souris est relachée");
}

void jButton1_actionPerformed(ActionEvent e) {
  jTextField1.setText("Toto");
 }
}
```

Attention sur un click de souris l'événement :

mouseClicked est **toujours** généré que le bouton soit **activé** :

ou bien **désactivé** :

Attention sur un click de souris l'événement :

actionPerformed est généré lorsque le bouton est **activé** :

actionPerformed **n'est pas** généré lorsque le bouton est **désactivé** :

Exemple - JcheckBox , JRadioButton

Objectif : Application simple Java utilisant deux objets de classe JCheckBox et JRadioButton.

cas d'un seul composant conteneur

La fenêtre d'exemple comporte 3 cases à cocher (jCheckBox1, jCheckBox2, jCheckBox3 : **JCheckBox**) et 3 boutons radios (jRadioButton1, jRadioButton2, jRadioButton3 : **JRadioButton**):

L'application dans cet exemple n'exécute aucune action (seul le click sur le composant est intéressant et se programme comme pour n'importe quel autre bouton de classe **JButton** par exemple). Nous observons seulement le comportement d'action en groupe en Java de ces boutons.

6 boutons ont été déposés sur la fenêtre (classe **JFrame** de type conteneur) :

Comme le montre l'image ci-haut, **tous** les radios boutons et les cases à cocher peuvent être **cochés en même temps** (contrairement au comportement des radios boutons de Delphi)

Cas de plus d'un composant conteneur

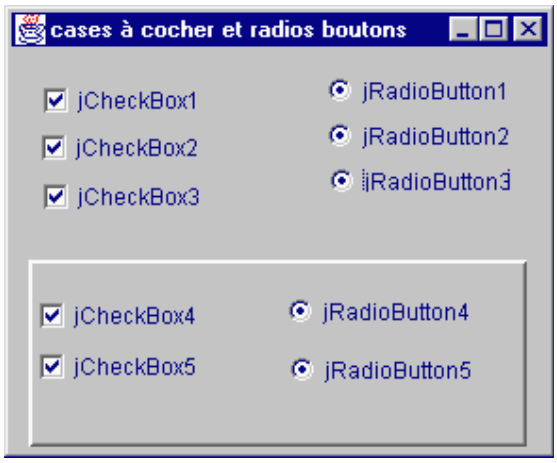
La fenêtre d'exemple comporte :

- 5 cases à cocher (jCheckBox1, jCheckBox2, jCheckBox3, jCheckBox4, jCheckBox5 : **JCheckBox**),
- 5 boutons radios (jRadioButton1, jRadioButton2, jRadioButton3, jRadioButton4, jRadioButton5 : **JRadioButton**),
- et un conteneur de type panneau (jPanel1 : **JPanel**).

jCheckBox1, jCheckBox2, jCheckBox3 sont déposés sur le conteneur fenêtre **JFrame**,
jRadioButton1, jRadioButton2, jRadioButton3 sont aussi déposés sur le conteneur fenêtre **JFrame**,

jCheckBox4, jCheckBox5 sont déposés sur le conteneur panneau **JPanel**,
jRadioButton4, jRadioButton5 sont déposés sur le conteneur panneau **JPanel**,

Voici le résultat obtenu :



Tous les composants peuvent être cochés, ils n'ont donc pas de comportement de groupe quel que soit le conteneur auquel ils appartiennent. IL faut en fait les rattacher à un groupe qui est représenté en Java par la classe ButtonGroup.

Regroupement de boutons s'excluant mutuellement

La classe ButtonGroup peut être utilisée avec n'importe quel genre d'objet dérivant de la classe **AbstractButton** comme les **JCheckBox**, **JRadioButton**, ***JRadioButtonMenuItem***, et aussi les ***JToggleButton*** et toute classe héritant de **AbstractButton** qui implémente une propriété de sélection.

Décidons dans l'exemple précédent de créer 2 objets de classe ButtonGroup que nous nommerons groupe1 et groupe2.

```
ButtonGroup Groupe1 = new ButtonGroup( );
ButtonGroup Groupe2 = new ButtonGroup( );
```

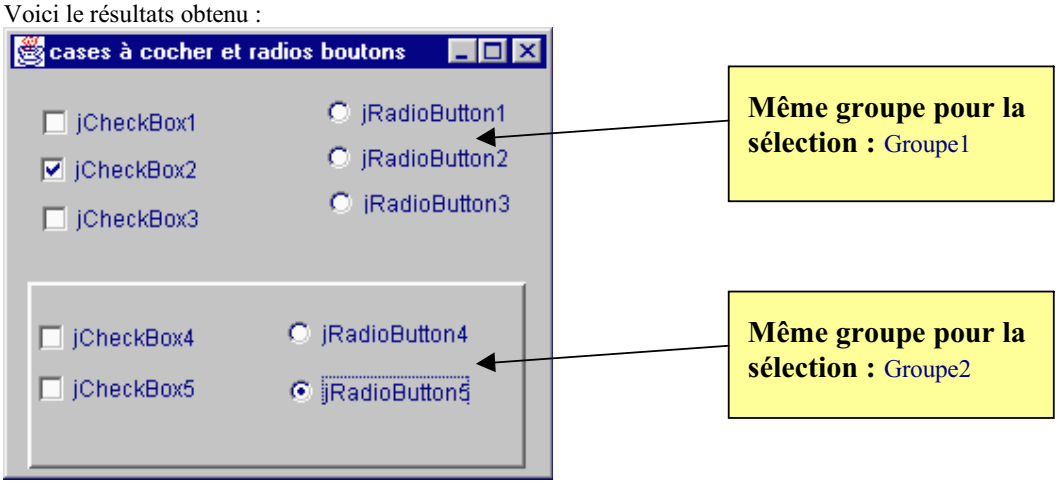
Nous rattachons au Groupe1 tous les composants déposés sur le **JPanel** (jCheckBox4, jCheckBox5, jRadioButton4 et jRadioButton5 font alors partie du même groupe d'exclusion mutuelle) :

```
Groupe1.add(jCheckBox1);
Groupe1.add(jCheckBox2);
Groupe1.add(jCheckBox3);
Groupe1.add(jRadioButton1);
Groupe1.add(jRadioButton2);
Groupe1.add(jRadioButton3);
```

Nous rattachons au Groupe2 tous les composants déposés sur le **JFrame** (jCheckBox1, jCheckBox2, jCheckBox3, jRadioButton1, jRadioButton2 et jRadioButton3 font donc partie d'un autre groupe d'exclusion mutuelle).

```
Groupe2.add(jCheckBox4);
Groupe2.add(jCheckBox5);
Groupe2.add(jRadioButton4);
Groupe2.add(jRadioButton5);
```

Voici le résultats obtenu :



Sur les 6 boutons déposés sur le JFrame seul un seul peut être coché, il en est de même pour les 4 boutons déposés sur le JPanel.

Amélioration interactive du JCheckBox sur le Checkbox

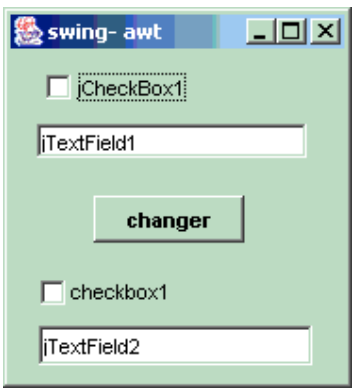
Il est possible dans le code, grâce à une méthode de modifier la valeur propriété cochée ou non cochée :

Avec un java.awt.Checkbox	Avec un javax.swing.JCheckBox
public void setState(boolean state)	**public void** setSelected(boolean b)

Dans les deux cas l'apparence visuelle de la case à cocher ne change pas (elle ne réagit qu'au click effectif de souris), ce qui limite considérablement l'intérêt d'utiliser une telle méthode puisque le visuel ne suit pas le programme.

Considérant cette inefficacité la bibliothèque swing propose une classe abstraite javax.swing.AbstractButton qui sert de modèle de construction à trois classes de composant JButton, JMenuItem, JtoggleButton et donc **JcheckBox** car celle-ci hérite de la classe JtoggleButton. Dans les membres que la classe javax.swing.AbstractButton offre à ses descendants existe la méthode doClick qui lance un click de souris utilisateur :

public void doClick()

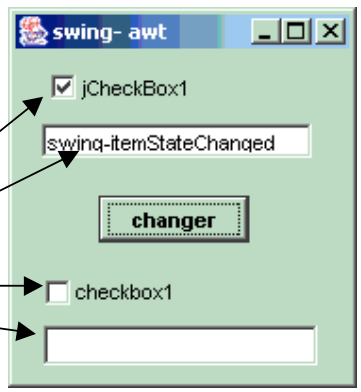


Code du click sur le bouton **changer** :

```
jCheckBox1.setSelected(false);
jCheckBox1.doClick();
checkbox1.setState(false);
```

Le jCheckBox1 a changé visuellement et a lancé son gestionnaire d'événement click.

Le Checkbox1 n'a pas changé visuellement et n'a pas lancé son gestionnaire d'événement click.



Exemple - JComboBox

Objectif : Application simple Java utilisant deux objets de classe JComboBox.

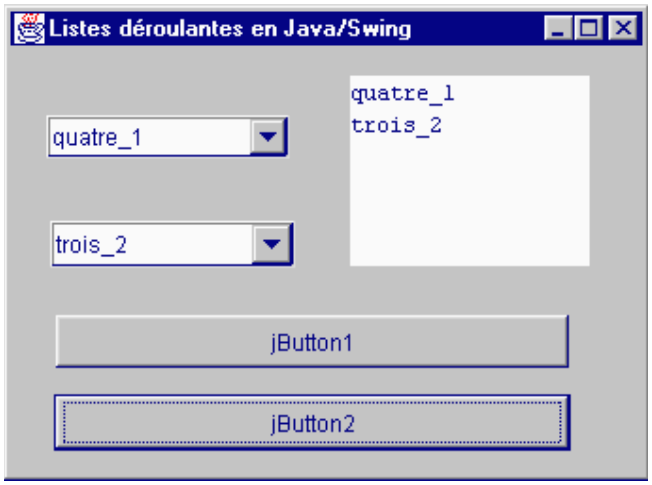
Dans cet exemple, nous utilisons deux JComboBox, le premier est chargé de données grâce à l'architecture MVC de la bibliothèque swing DefaultComboBoxModel, *le second est chargé de données directement à travers sa méthode* addItem .

La fenêtre comporte :

deux boutons (**JButton** jButton1 et jButton2),

deux listes déroulantes (**JComboBox** jComboBox1 et jComboBox2),

et un éditeur de texte multi-ligne (**JTextArea** jTextArea1)



Ci-contre le diagramme événementiel de l'action du click de souris sur le bouton jButton1:

lorsqu'un élément de la liste est sélectionné lors du click sur le bouton, l'application rajoute cet élément dans la zone de texte jTextArea1.

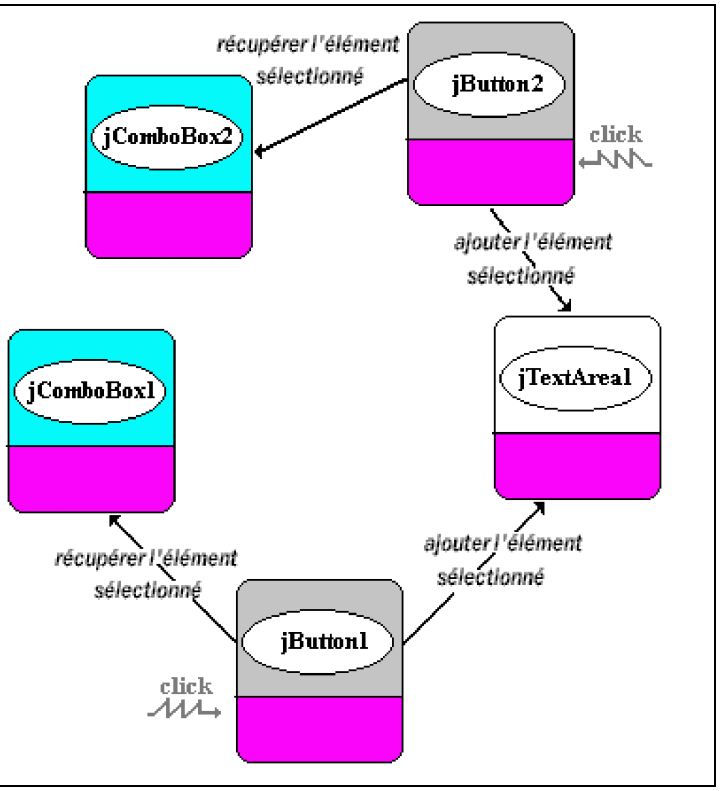
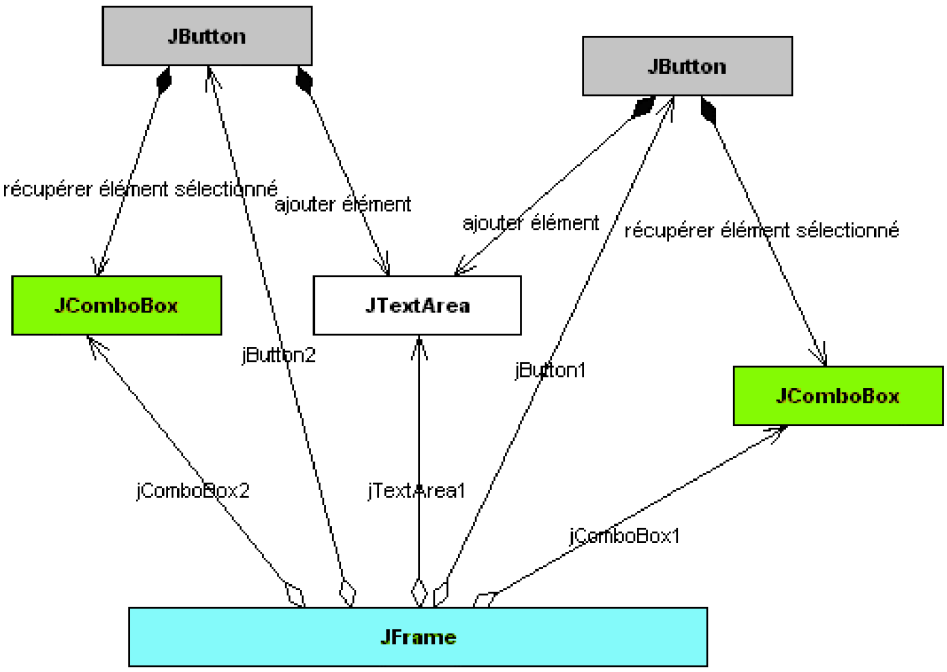
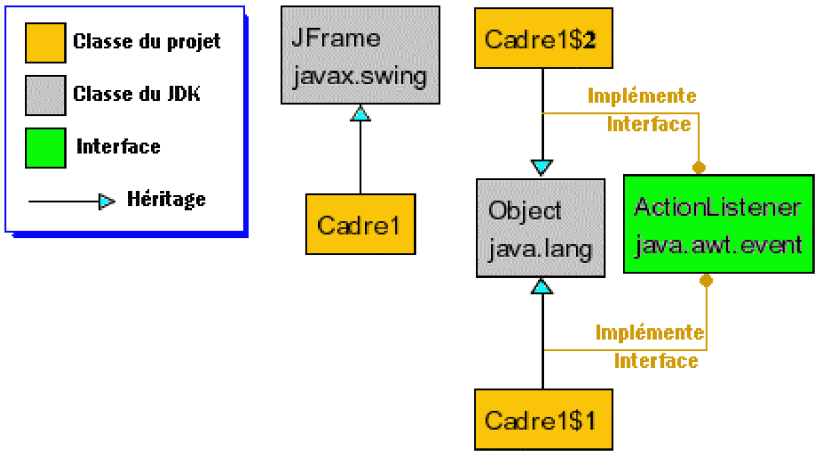


Schéma UML du projet



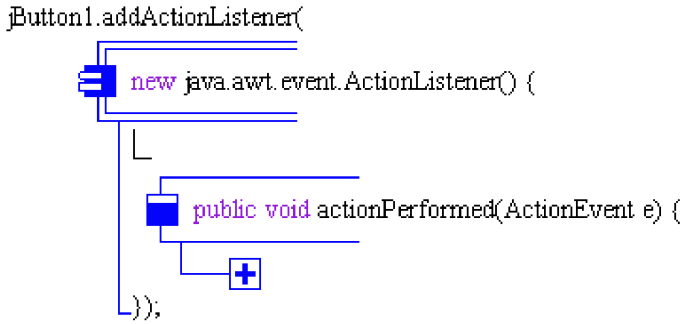
En Java (génération du code source effectuée par JBuilder)

Comme en java tous les événements sont interceptés par des objets écouteurs, ci-dessous nous donnons les diagrammes UML des classes utilisées par le programme qui est proposé :

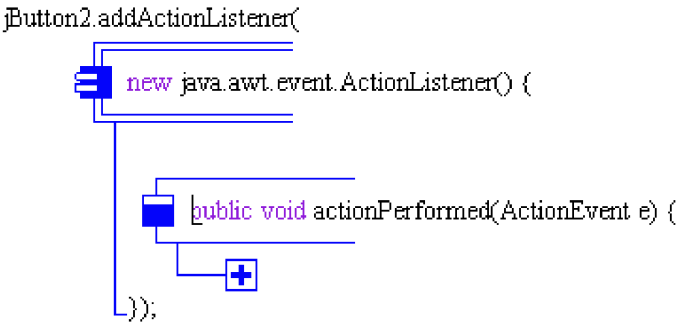


Rappelons que les classes Cadre1$1 et Cadre1$2 sont des classes anonymes créées lors de la déclaration de l'écouteur des boutons jButton1 et jButton2, Java 2 crée donc dynamiquement un objet écouteur interne (dont la référence n'est pas disponible). Ci-dessous les diagrammes jGrasp des classes anonymes cadre1$1 et Cadre1$2 :

Cadre1$1:



Cadre1$2:



Enfin pour terminer, voici le listing Java/Swing complet de la classe représentant la fenêtre :

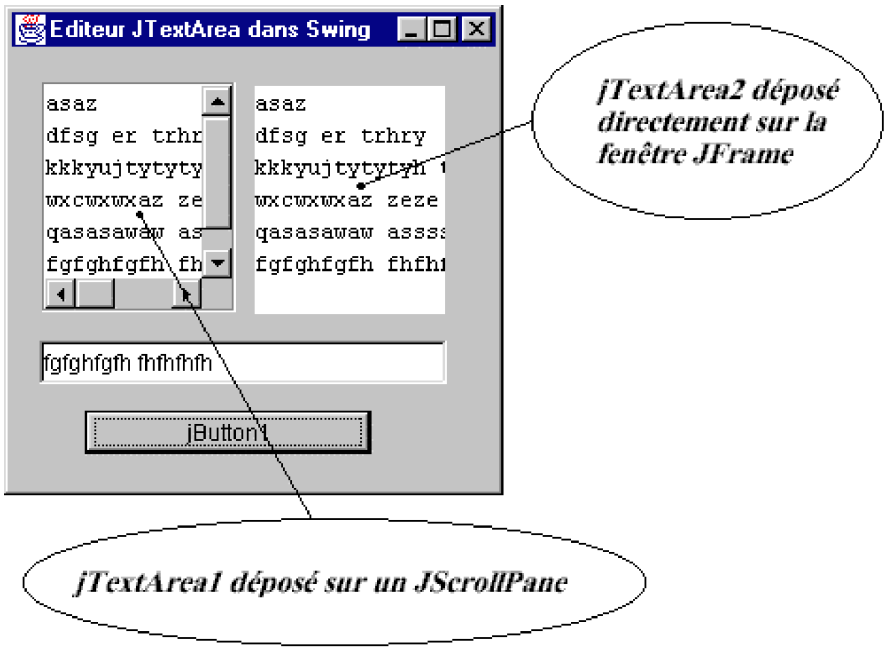
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Cadre1 extends JFrame {
  DefaultComboBoxModel mdc = new DefaultComboBoxModel();
  JComboBox jComboBox2 = new JComboBox();
  JComboBox jComboBox1 = new JComboBox();
  JTextArea jTextArea1 = new JTextArea();
  JButton jButton1 = new JButton();
  JButton jButton2 = new JButton();

   //Construire le cadre
  public Cadre1() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try  {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }

  //Initialiser le composant
  private void jbInit() throws Exception  {
    this.getContentPane().setLayout(null);
    this.setSize(new Dimension(343, 253));
    this.setTitle("Listes déroulantes en Java/Swing");
    jTextArea1.setBounds(new Rectangle(180, 15, 127, 101));
```

```
    jButton1.setText("jButton1");
    jButton1.setBounds(new Rectangle(24, 142, 272, 28));

      // Ecouteur de jButton1 en classe anonyme :
    jButton1.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        jButton1_actionPerformed(e);
      }
    });

    jComboBox2.setBounds(new Rectangle(21, 92, 130, 25));
    jComboBox1.setBounds(new Rectangle(19, 36, 129, 23));
    jComboBox1.setModel(mdc); // la première liste déroulante est dirigée par son modèle MVC
    jButton2.setText("jButton2");
    jButton2.setBounds(new Rectangle(23, 184, 274, 30));

    // Ecouteur de jButton2 en classe anonyme :
    jButton2.addActionListener(new java.awt.event.ActionListener() {

      public void actionPerformed(ActionEvent e) {
        jButton2_actionPerformed(e);
      }
    });

    this.getContentPane().add(jComboBox1, null);
    this.getContentPane().add(jComboBox2, null);
    this.getContentPane().add(jTextArea1, null);
    this.getContentPane().add(jButton2, null);
    this.getContentPane().add(jButton1, null);

    // Le modèle MVC de la première liste déroulante :
    mdc.addElement("un_1");
    mdc.addElement("deux_1");
    mdc.addElement("trois_1");
    mdc.addElement("quatre_1");

     // La seconde liste déroulante est chargée directement :
    jComboBox2.addItem("un_2");
    jComboBox2.addItem("deux_2");
    jComboBox2.addItem("trois_2");
    jComboBox2.addItem("quatre_2");
  }

  //Remplacé (surchargé) pour pouvoir quitter lors de System Close
  protected void processWindowEvent(WindowEvent e) {
    super.processWindowEvent(e);
    if(e.getID() == WindowEvent.WINDOW_CLOSING) {
      System.exit(0);
    }
  }

  void jButton1_actionPerformed(ActionEvent e) {
    jTextArea1.append((String)jComboBox1.getSelectedItem()+"\n");
  }

  void jButton2_actionPerformed(ActionEvent e) {
    jTextArea1.append((String)jComboBox2.getSelectedItem()+"\n");
  }
}
```

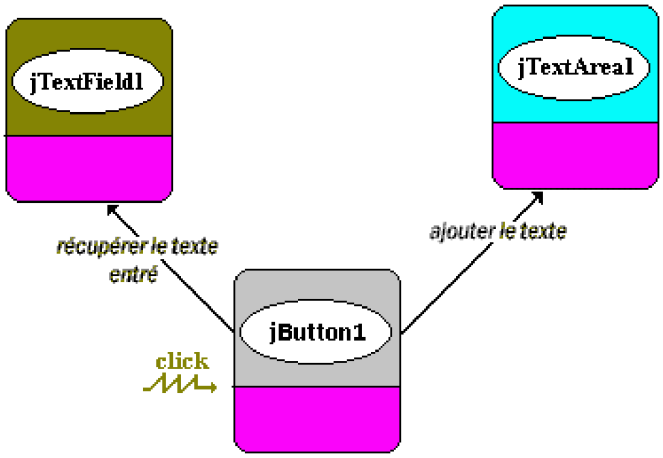
Exemple - JTextArea

Objectif : Application simple Java utilisant deux objets de classe JTextArea.

La fenêtre comporte un bouton (**JButton** jButton1), un éditeur mono-ligne(**JTextField** jTextField1) et deux éditeurs de texte multi-lignes (**JTextArea** jTextArea1, jTextArea2):



L'application consiste après qu'un texte ait été entré dans le jTextField1, sur le clic du bouton jButton1 à déclencher l'ajout de ce texte dans jTextArea1 (éditeur de gauche).



Le **JTextArea** délègue la gestion des barres de défilement à un objet conteneur dont c'est la fonction le ***JScrollpane***, ceci a lieu lorsque le **JTextArea** est ajouté au ***JScrollpane.***

Gestion des barres de défilement du texte

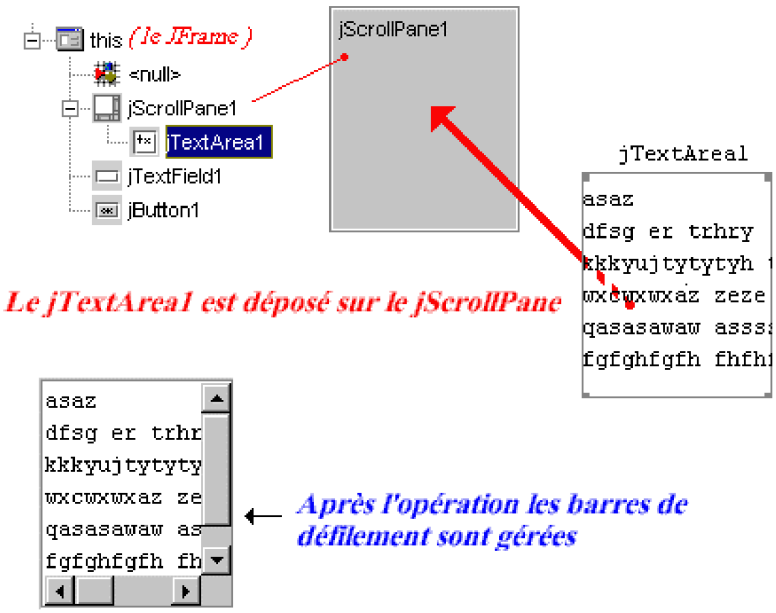
Afin de bien comprendre la gestion des barres de défilement verticales et horizontales, le programme ci-dessous contient deux objets jTextArea1 et jTextArea2 dont le premier est déposé dans un objet conteneur de classe ***JScrollpane*** (classe encapsulant la gestion de barres de défilements), selon l'un des deux codes sources suivants :

```
// les déclarations :
JScrollPane jScrollPane1 = new JScrollPane( );
JTextArea jTextArea1 = new JTextArea( );

// dans le constructeur de la fenêtre JFrame :
this.getContentPane( ).add(jScrollPane1, null);
jScrollPane1.getViewport( ).add(jTextArea1, null);
```

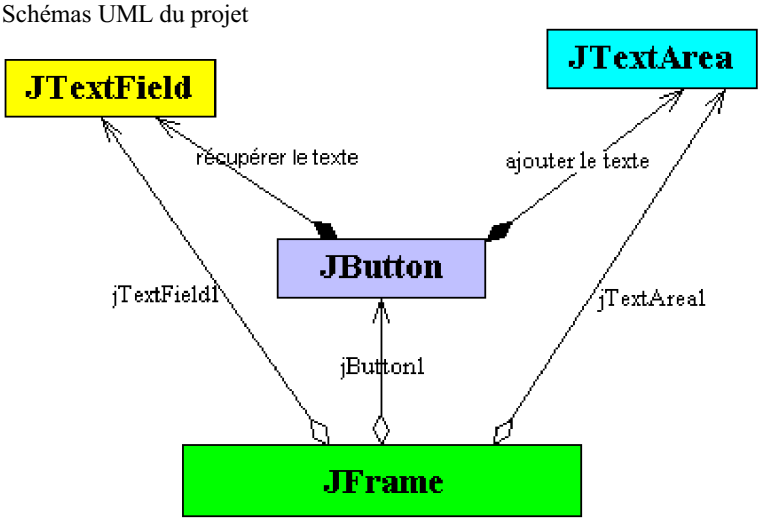
ou bien en utilisant un autre constructeur de JScrollPane :

```
// les déclarations :
JTextArea jTextArea1 = new JTextArea( );
JScrollPane jScrollPane1 = new JScrollPane( jTextArea1 );

// dans le constructeur de la fenêtre JFrame :
this.getContentPane( ).add(jScrollPane1, null);
```

Voici en résumé ce qu'il se passe lors de l'exécution de ce code (visualisation avec un RAD permettant d'avoir un explorateur de classes et de conteneur conseillée, ici le traitement est effectué avec JBuilder) :

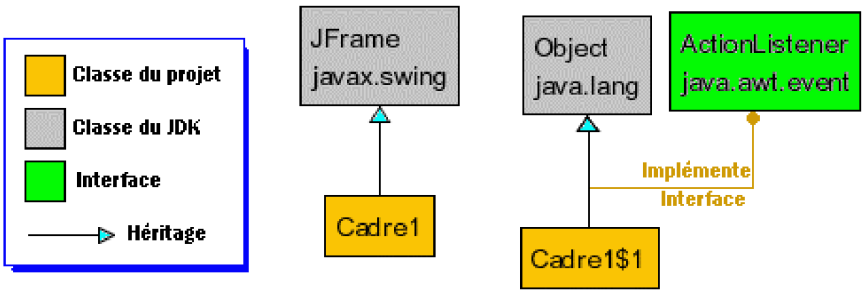


Schémas UML du projet



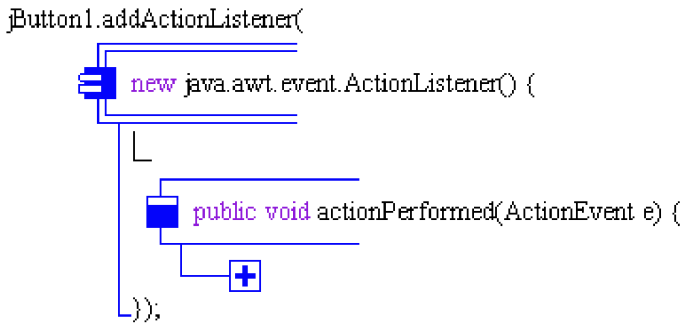
En Java (génération du code source effectuée par JBuilder)

Comme en java tous les événements sont interceptés par des objets écouteurs, ci-dessous nous donnons les diagrammes UML des classes utilisées par le programme qui est proposé :



Rappelons que la classe Cadre1$1 est une classe anonyme créée lors de la déclaration de l'écouteur du bouton jButton1, Java 2 crée donc dynamiquement un objet écouteur interne (dont la référence n'est pas disponible). Ci-dessous le diagramme jGrasp de la classe anonyme cadre1$1

Cadre1$1:



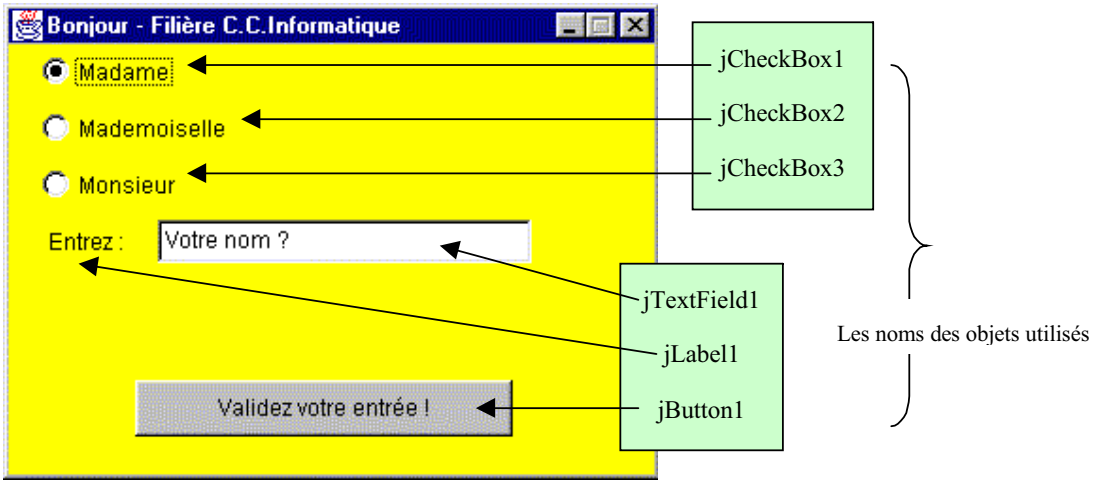
Enfin pour terminer, voici le listing Java/Swing complet de la classe représentant la fenêtre (y compris le deuxième JTextArea de vérification) :

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Cadre1 extends JFrame {
  JTextField jTextField1 = new JTextField();
  JButton jButton1 = new JButton();
  JScrollPane jScrollPane1 = new JScrollPane();
  JTextArea jTextArea2 = new JTextArea();
  JTextArea jTextArea1 = new JTextArea();

   //Construire le cadre
  public Cadre1() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try  {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }

  //Initialiser le composant
  private void jbInit() throws Exception  {
    this.getContentPane().setLayout(null);
    this.setSize(new Dimension(265, 260));
    this.setTitle("Editeur JTextArea dans Swing");
    jTextField1.setText("jTextField1");
    jTextField1.setBounds(new Rectangle(15, 155, 216, 23));
    jButton1.setText("jButton1");
    jButton1.setBounds(new Rectangle(39, 192, 152, 23));
    jButton1.addActionListener(new java.awt.event.ActionListener() {

      public void actionPerformed(ActionEvent e) {
        jButton1_actionPerformed(e);
      }
    });
    jScrollPane1.setBorder(null);
    jScrollPane1.setBounds(new Rectangle(16, 18, 103, 122));
    jTextArea2.setText("jTextArea2");
    jTextArea2.setBounds(new Rectangle(129, 20, 101, 121));
    jTextArea1.setText("jTextArea1");
    this.getContentPane().add(jScrollPane1, null);
    jScrollPane1.getViewport().add(jTextArea1, null);
    this.getContentPane().add(jTextArea2, null);
    this.getContentPane().add(jTextField1, null);
    this.getContentPane().add(jButton1, null);
  }

 //Remplacé (surchargé) pour pouvoir quitter lors de System Close
  protected void processWindowEvent(WindowEvent e) {
    super.processWindowEvent(e);
    if(e.getID() == WindowEvent.WINDOW_CLOSING) {
      System.exit(0);
    }
  }
```

```
  void jButton1_actionPerformed(ActionEvent e) {
    jTextArea1.append(jTextField1.getText()+"\n");
    jTextArea2.append(jTextField1.getText()+"\n"); // copie pour vérification visuelle
  }
}
```

Saisie interactive de renseignements avec des swing

Nous reprenons l'IHM de saisie de renseignements concernant un(e) étudiant(e) que nous avons déjà construite sans événement, rajoutons des événements pour la rendre interactive, elle stockera les renseignements saisis dans un fichier de texte éditable avec un quelconque traitement de texte :



Description événementielle de l'IHM :

- Dans l'IHM au départ le **jButton1** est désactivé, aucun **jCheckBox** n'est coché, le **jTextField1** est vide.
- Dès que l'un des **jCheckBox** est coché, et que le **jTextField1** contient du texte le **jButton1** est activé, dans le cas contraire le **jButton1** est désactivé (dès que le **jTextField1** est vide).
- Un click sur le **jButton1** sauvegarde les informations dans le fichier texte **etudiants.txt**.
- La fiche qui dérive d'une **JFrame** se ferme et arrête l'application sur le click du bouton de fermeture.

```
import java.awt.*; // utilisation des classes du package awt
import java.awt.event.*; // utilisation des classes du package awt
import java.io.*; // utilisation des classes du package io
import javax.swing.*; // utilisation des classes du package swing
import java.util.*; // utilisation des classes du package util pour Enumeration
import javax.swing.text.*;// utilisation pour Document
import javax.swing.event.*;// utilisation pour DocumentEvent

 class ficheSaisie extends JFrame { // la classe ficheSaisie hérite de la classe des fenêtres JFrame
   JButton jbutton1 = new JButton( );// création d'un objet de classe JButton
   JLabel jlabel1 = new JLabel( );// création d'un objet de classe JLabel
   ButtonGroup Group1 = new ButtonGroup ( );// création d'un objet groupe pour AbstractButton et dérivées
```

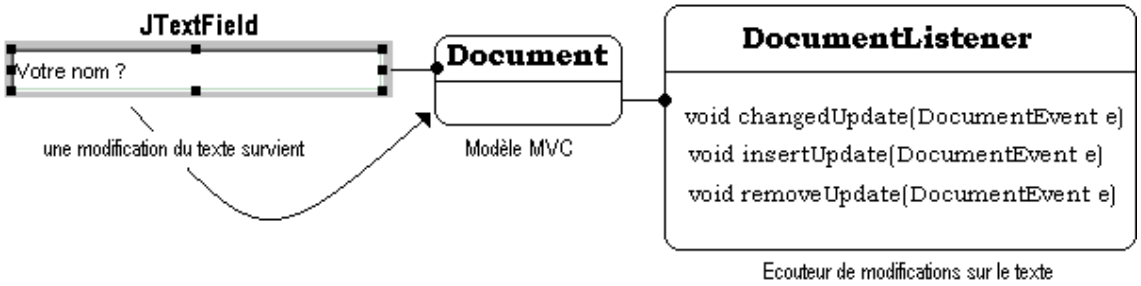
```
   JCheckBox jcheckbox1 = new JCheckBox( );// création d'un objet de classe JCheckbox
   JCheckBox jcheckbox2 = new JCheckBox( );// création d'un objet de classe JCheckbox
   JCheckBox jcheckbox3 = new JCheckBox( );// création d'un objet de classe JCheckbox
   JTextField jtextField1 = new JTextField( );// création d'un objet de classe JTextField
   Container ContentPane;

   private String EtatCivil;//champ = le label du checkbox coché
   private FileWriter fluxwrite; //flux en écriture (fichier texte)
   private  BufferedWriter fluxout;//tampon pour lignes du fichier


//Constructeur de la fenêtre
    public ficheSaisie ( ) {  //Constructeur sans paramètre
      Initialiser( );// Appel à une méthode privée de la classe
   }

//indique quel JCheckBox est coché(réf) ou si aucun n'est coché(null)  :
    private JCheckBox getSelectedjCheckbox(){
      JCheckBox isSelect=null;
      Enumeration checkBenum = Group1.getElements();
      for (int i = 0; i < Group1.getButtonCount(); i++) {
        JCheckBox B =(JCheckBox) checkBenum.nextElement();
        if(B.isSelected())
          isSelect = B;
      }
      return isSelect;
   }
//Active ou désactive le bouton pour sauvegarde :
    private void AutoriserSave(){
      if (jtextField1.getText().length() !=0 &&  this.getSelectedjCheckbox()!=null)
        jbutton1.setEnabled(true);
      else
        jbutton1.setEnabled(false);
   }
//rempli le champ Etatcivil selon le checkBox coché :
    private void StoreEtatcivil(){
      this.AutoriserSave();
      if (this.getSelectedjCheckbox()!=null)
        this.EtatCivil=this.getSelectedjCheckbox().getLabel();
      else
        this.EtatCivil="";
   }
//sauvegarde les infos étudiants dans le fichier :
    public void ecrireEnreg(String record) {
      try {
        fluxout.write(record);//écrit les infos
        fluxout.newLine( ); //écrit le eoln
      }
         catch (IOException err) {
           System.out.println( "Erreur : " + err );      }
   }

//Initialiser la fenêtre :
    private void Initialiser( ) {  //Création et positionnement de tous les composants
      ContentPane = this.getContentPane( ) ; //Référencement du fond de dépôt des composants
      this.setResizable(false); // la fenêtre ne peut pas être retaillée par l'utilisateur
      ContentPane.setLayout(null); // pas de Layout, nous positionnons les composants nous-mêmes
      ContentPane.setBackground(Color.yellow); // couleur du fond de la fenêtre
      this.setSize(348, 253); // width et height de la fenêtre
      this.setTitle("Bonjour - Filière C.C.Informatique"); // titre de la fenêtre
```

```
this.setForeground(Color.black); // couleur de premier plan de la fenêtre
jbutton1.setBounds(70, 160, 200, 30); // positionnement du bouton
jbutton1.setText("Validez votre entrée !"); // titre du bouton
jbutton1.setEnabled(false); // bouton désactivé
jlabel1.setBounds(24, 115, 50, 23); // positionnement de l'étiquette
jlabel1.setText("Entrez :"); // titre de l'étiquette
jcheckbox1.setBounds(20, 25, 88, 23); // positionnement du JCheckBox
Group1.add(jcheckbox1); // ce JCheckBox est mis dans le groupe Group1
jcheckbox1.setText("Madame");// titre du JCheckBox
jcheckbox2.setBounds(20, 55, 108, 23);// positionnement du JCheckBox
Group1.add(jcheckbox2); // ce JCheckBox est mis dans le groupe Group1
jcheckbox2.setText("Mademoiselle");// titre du JCheckBox
jcheckbox3.setBounds(20, 85, 88, 23);// positionnement du JCheckBox
Group1.add(jcheckbox3); // ce JCheckBox est mis dans le groupe Group1
jcheckbox3.setText("Monsieur");// titre du JCheckBox
jcheckbox1.setBackground(Color.yellow);//couleur du fond du jcheckbox1
jcheckbox2.setBackground(Color.yellow);//couleur du fond du jcheckbox2
jcheckbox3.setBackground(Color.yellow);//couleur du fond du jcheckbox3
jtextField1.setBackground(Color.white);// couleur du fond de l'éditeur mono ligne
jtextField1.setBounds(82, 115, 198, 23);// positionnement de l'éditeur mono ligne
jtextField1.setText("Votre nom ?");// texte de départ de l'éditeur mono ligne
ContentPane.add(jcheckbox1);// ajout dans la fenêtre du JCheckBox
ContentPane.add(jcheckbox2);// ajout dans le ContentPane  de la fenêtre du JCheckBox
ContentPane.add(jcheckbox3);// ajout dans le ContentPane  de la fenêtre du JCheckBox
ContentPane.add(jbutton1);// ajout dans le ContentPane  de la fenêtre du bouton
ContentPane.add(jtextField1);// ajout dans le ContentPane  de la fenêtre de l'éditeur mono ligne
ContentPane.add(jlabel1);// ajout dans le ContentPane  de la fenêtre de l'étiquette
EtatCivil ="";// pas encore de valeur
Document doc = jtextField1.getDocument();// partie Modele MVC du JTextField
try{
  fluxwrite = new FileWriter("etudiants.txt",true);// création du fichier(en mode ajout)
  fluxout = new BufferedWriter(fluxwrite); //tampon de ligne associé
}
   catch(IOException err){ System.out.println( "Problème dans l'ouverture du fichier ");}
//--> événements et écouteurs :
this.addWindowListener(
     new WindowAdapter(){
       public void windowClosing(WindowEvent e){
         try{
           fluxout.close( ); //le fichier est fermé et le tampon vidé
         }
           catch(IOException err){ System.out.println( "Impossible de fermer le fichier ");}
         System.exit(100);
       }
     });
doc.addDocumentListener(//on crée un écouteur anonymepour le  Modele (qui fait aussi le controle)
     new javax.swing.event.DocumentListener() {
      //-- l'interface DocumentListener a 3 méthodes qu'il faut implémenter :
       public void changedUpdate(DocumentEvent e) {}

       public void removeUpdate(DocumentEvent e) {//une suppression est un changement de texte
         textValueChanged(e);// appel au gestionnaire de changement de texte
       }

       public void insertUpdate(DocumentEvent e) {//une insertion est un changement de texte
         textValueChanged(e);// appel au gestionnaire de changement de texte
       }
     });
jcheckbox1.addItemListener(
     new ItemListener(){
```

```
            public void itemStateChanged(ItemEvent e){
              StoreEtatcivil( );
            }
          });
      jcheckbox2.addItemListener(
          new ItemListener(){
            public void itemStateChanged(ItemEvent e){
              StoreEtatcivil();
            }
          });
      jcheckbox3.addItemListener(
          new ItemListener(){
            public void itemStateChanged(ItemEvent e){
              StoreEtatcivil();
            }
          });
      jbutton1.addActionListener(
          new java.awt.event.ActionListener() {
            public void actionPerformed(ActionEvent e) {
              ecrireEnreg(EtatCivil+":"+jtextField1.getText());
            }
          });
    }
    //Gestionnaire du changement de texte dans un document
     private void textValueChanged(DocumentEvent e) {
      AutoriserSave();
    }
  }
```

Remarques: (comparez ce code au même exercice construit en Awt et notez les différences)

- Un JTextField n'est pas comme un TextField directement sensible au changement de son texte, c'est son modèle MVC du type Document qui assure la gestion et la réaction à la survenue de modifications du texte en recensant des écouteurs de classe héritant de l'interface DocumentListener :



Un **ButtonGroup** ne possède pas comme un **CheckboxGroup** Awt , de méthode **getSelectedjCheckbox** permettant de connaître le bouton du groupe qui est coché. Nous avons construit une telle méthode **getSelectedjCheckbox** qui renvoie la référence d'un **JCheckBox** semblablement à la méthode des Awt :

```
  //-- indique quel JCheckBox est coché(réf) ou si aucun n'est coché(null)  :
   private JCheckBox getSelectedjCheckbox(){
    JCheckBox isSelect=null;
    Enumeration checkBenum = Group1.getElements();
    for (int i = 0; i < Group1.getButtonCount(); i++) {
      JCheckBox B =(JCheckBox) checkBenum.nextElement();
      if(B.isSelected())
        isSelect = B;
    }
    return isSelect;
  }
```

Les applets Java : applications internet

La page HTML minimale

Jusqu'à présent tous les programmes Java qui ont été écrits dans les autres chapitres sont des applications autonomes pouvant fonctionner directement sur une plateforme Windows, Unix, Linux, MacOs...

Une des raisons initiales, du succès de Java peut être la raison majeure, réside dans la capacité de ce langage à créer des applications directement exécutables dans un navigateur contenant une machine virtuelle java. Ces applications doivent être insérées dans le code interne d'une page HTML (**H**yper**T**ext **M**arkup **L**anguage) entre deux balises spécifiques.

Ces applications insérées dans une page HTML, peuvent être exécutées en local sur la machine hôte, le navigateur jouant le rôle d'environnement d'exécution. Elles peuvent aussi être exécutées en local par votre navigateur pendant une connexion à internet par téléchargement de l'application en même temps que le code HTML de la page se charge. Ces applications Java non autonomes sont dénommées des **applets**.

Applet = **Appl**ication Intern**et** écrite en Java intégrée à une page HTML ne pouvant être qu'exécutée par un navigateur et s'affichant dans la page HTML.

Une page HTML est un fichier texte contenant des descriptions de la mise en page du texte et des images. Ces descriptions sont destinées au navigateur afin qu'il assure l'affichage de la page, elles s'effectuent par l'intermédiare de balises (parenthèses de description contextuelles). Un fichier HTML commence toujours par la balise <HTML> et termine par la balise </HTML>.

Une page HTML minimale contient deux sections :

- l'en-tête de la page balisée par <HEAD> ...</HEAD>
- le corps de la page balisé par <BODY> ... </BODY>

Voici un fichier texte minimal pour une page HTML :

```
<HTML>
    <HEAD>


    </HEAD>
    <BODY>
    </BODY>
```

</HTML>

Une applet Java est invoquée grâce à deux balises spéciales insérées au sein du corps de la page HTML :

```
<APPLET CODE =.....>
</APPLET>
```

Nous reverrons plus loin la signification des paramètres internes à la balise <APPLET>.

Terminons cette introduction en signalant qu'une applet, pour des raisons de sécurité, est soumise à certaines restrictions notamment en matière d'accès aux disques durs de la machine hôte.

La classe java.applet.Applet

Cette classe joue un rôle semblable à celui de la classe Frame dans le cas d'une application. Une Applet est un Panel spécial qui se dessinera sur le fond de la page HTML dans laquelle il est inséré.

Voici la hiérarchie des classes dont Applet dérive :

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--java.awt.Panel
                    |
                    +--java.applet.Applet
```

Etant donné l'importance de la classe, nous livrons ci-dessous, la documentation du JDK de la classe Applet.

Minimum requis pour une applet

Si vous voulez écrire des applets, il faut posséder un navigateur pour exécuter les applet et un éditeur de texte permettant d'écrire le code source Java.

Vous pouvez aussi utiliser un environnement de développement Java

Tout environnement de développement java doit contenir un moyen de visionner le résultat de la programmation de votre applet, cet environnement fera appel à une visionneuse d'applet (dénommée AppletViewer dans le JDK).

Votre applet doit hériter de la classe Applet

public class AppletExemple **extends** Applet { .....
}
Comme toutes les classes Java exécutables, le nom du fichier doit correspondre très exactement au nom de la classe avec comme suffixe java (ici : **AppletExemple.java**)

Constructor Summary

Applet()

Method Summary

void	**destroy**() Called by the browser or applet viewer to inform this applet that it is being reclaimed and that it should destroy any resources that it has allocated.
AppletContext	**getAppletContext**() Determines this applet's context, which allows the applet to query and affect the environment in which it runs.
String	**getAppletInfo**() Returns information about this applet.
AudioClip	**getAudioClip**(URL url) Returns the AudioClip object specified by the URL argument.
AudioClip	**getAudioClip**(URL url, String name) Returns the AudioClip object specified by the URL and name arguments.
URL	**getCodeBase**() Gets the base URL.
URL	**getDocumentBase**() Gets the document URL.
Image	**getImage**(URL url) Returns an Image object that can then be painted on the screen.
Image	**getImage**(URL url, String name) Returns an Image object that can then be painted on the screen.

Locale	**getLocale**() Gets the Locale for the applet, if it has been set.
String	**getParameter**(String name) Returns the value of the named parameter in the HTML tag.
String[][]	**getParameterInfo**() Returns information about the parameters than are understood by this applet.
void	**init**() Called by the browser or applet viewer to inform this applet that it has been loaded into the system.
boolean	**isActive**() Determines if this applet is active.
static AudioClip	**newAudioClip**(URL url) Get an audio clip from the given URL
void	**play**(URL url) Plays the audio clip at the specified absolute URL.
void	**play**(URL url, String name) Plays the audio clip given the URL and a specifier that is relative to it.
void	**resize**(Dimension d) Requests that this applet be resized.
void	**resize**(int width, int height) Requests that this applet be resized.
void	**setStub**(AppletStub stub) Sets this applet's stub.
void	**showStatus**(String msg) Requests that the argument string be displayed in the "status window".
void	**start**() Called by the browser or applet viewer to inform this applet that it should start its execution.
void	**stop**() Called by the browser or applet viewer to inform this applet that it should stop its execution.

Fonctionnement d'une applet

Nous allons examiner quelques unes des 23 méthodes de la classe Applet, essentielles à la construction et à l'utilisation d'une applet.

La méthode init( )

Lorsqu'une applet s'exécute la méthode principale qui s'appelait **main( )** dans le cas d'une application Java se dénomme ici **init( )**. Il nous appartient donc de **surcharger dynamiquement**

(redéfinir) la méthode **init** de la classe Applet afin que notre applet fonctionne selon nos attentes.

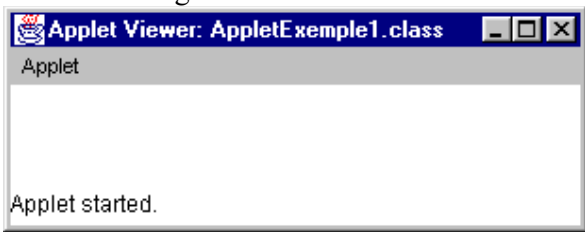
La méthode **init( )** est appelée **une seule fois**, lors du chargement de l'applet avant que celui-ci ne soit affiché.

Exemple d'applet vide :

Code Java

```
import java.applet.*;
public class AppletExemple1 extends Applet  {


}
```

Voici l'affichage obtenu :



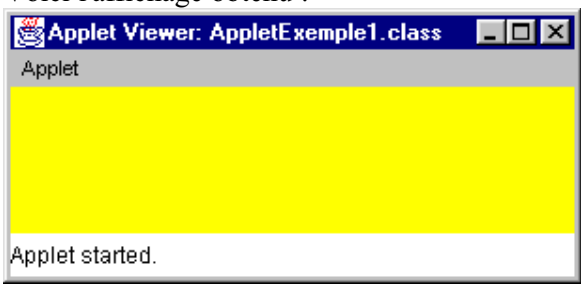
Soit à colorier le fond de l'applet, nous allons programmer le changement du fond de l'objet applet à partir de la méthode **init** de l'applet (avant même qu'il ne s'affiche).

```
public void init( ) {
 // avant le démarrage de l'affichage de l'applet
 this.setBackground(Color.yellow); // équivalent à : setBackground(Color.yellow);
}
```

Code Java

```
import java.applet.*;
public class AppletExemple1 extends Applet
{  public void init( ) {
     this.setBackground(Color.yellow);
   }
}
```

Voici l'affichage obtenu :



La méthode start( )

La méthode **start( )** est appelée **à chaque fois**, soit :

- après la méthode init
- après chaque modification de la taille de l'applet (minimisation,...).

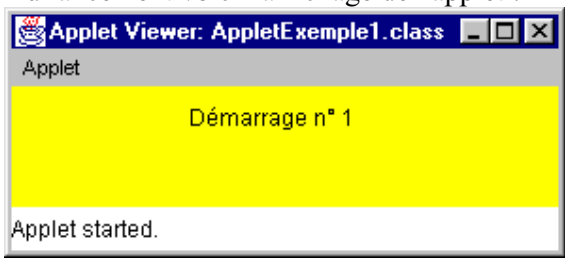
Si nous voulons que notre applet effectue une action spécifique au redémarrage, il suffit de **surcharger dynamiquement (redéfinir)** la méthode **start( )**.

Soit à compter et afficher dans une Label, le nombre d'appels à la méthode start :

Code Java

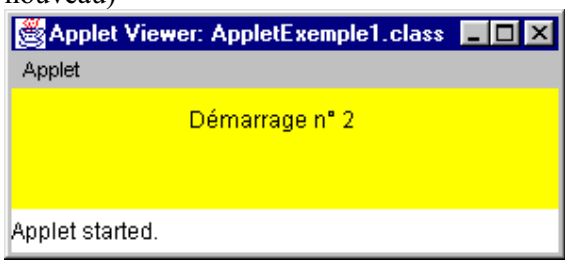
```
import java.applet.*;
public class AppletExemple1 extends Applet
{   Label etiq = new Label("Démarrage n&deg; ");
    int nbrDemarr = 0 ;
    public void init( ) {
    this.setBackground(Color.yellow);
    this.add(etiq);
   }
   public void start( ) {
     nbrDemarr++;
     etiq.setText( "Démarrage n&deg; "+String.valueOf ( nbrDemarr ) );
   }
}
```

Au lancement voici l'affichage de l'applet :



On minimise l'applet dans la bare des tâches :

On restaure l'applet à partir de la barre des tâches : (**start** est appelée automatiquemen à nouveau)



etc....

La méthode paint( )

La méthode **paint(Graphics x)** est appelée **à chaque fois**, soit :

- après que l'applet a été masquée, déplacée, retaillée,...
- à chaque réaffichage de l'applet (après minimisation,...).

Cette méthode est chargée de l'affichage de tout ce qui est graphique, vous mettrez dans le corps de cette méthode votre code d'affichage de vos éléments graphiques afin qu'ils soient redessinés systématiquement.

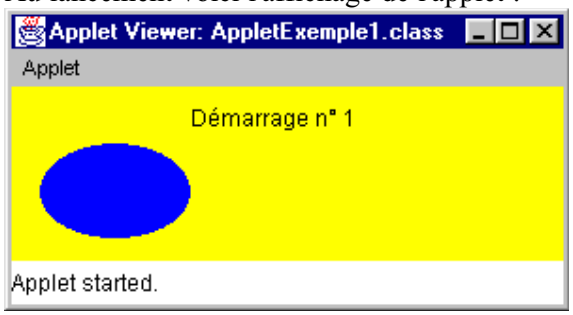
Si nous voulons que notre applet redessine nos graphiques, il suffit de **surcharger dynamiquement (redéfinir)** la méthode **paint(Graphics x)**.

Soit à dessiner dans l'applet précédent une ellipse peinte en bleu, on rajoute le code dans la méthode paint :

Code Java

```
import java.applet.*;
public class AppletExemple1 extends Applet
{   Label etiq = new Label("Démarrage n&deg; ");
    int nbrDemarr = 0 ;
    public void init( ) {
    this.setBackground(Color.yellow);
    this.add(etiq);
   }
   public void start( ) {
     nbrDemarr++;
     etiq.setText( "Démarrage n&deg; "+String.valueOf ( nbrDemarr ) );
   }
  public void paint (Graphics x) {
    this.setForeground(Color.blue);
    x.fillOval(15,30,80,50);
  }
}
```

Au lancement voici l'affichage de l'applet :



Les méthodes stop( ) et destroy( )

La méthode **stop( )** est appelée **à chaque fois**, soit :

- que l'applet a été masquée dans la page du navigateur (défilement vertical ou horizontal dans le navigateur), déplacée, retaillée,...
- lors de l'abandon et du changement de la page dans le navigateur.

Cette méthode arrête toutes les actions en cours de l'applet.

Si nous voulons que notre applet effectue des actions spécifiques lors de son arrêt (comme désactiver ou endormir des Threads, envoyer des messages...), il suffit de **surcharger dynamiquement (redéfinir)** la méthode **stop( )**.

La méthode **destroy( )** est appelée **à chaque fois**, soit :

- que l'utilisateur charge une nouvelle page dans le navigateur
- lors de la fermeture du navigateur

Pour mémoire la méthode **destroy( )** est invoquée pour libérer toutes les ressources que l'applet utilisait, normalement la machine virtuelle Java se charge de récupérer automatiquement la mémoire.

Si nous voulons que notre applet effectue des actions spécifiques lors de son arrêt (comme terminer définitivement des Threads), il suffit de **surcharger dynamiquement (redéfinir)** la méthode **destroy( )**.

Une applet dans une page HTML

Le code d'appel d'une applet

Nous avons déjà indiqué au premier paragraphe de ce chapitre que le code d'appel d'une applet est intégré au texte source de la page dans laquelle l'applet va être affichée. Ce sont les balises <APPLET CODE =.....> et </APPLET> qui précisent les modalités de fonctionnement de l'applet. En outre, l'applet s'affichera dans la page exactement à l'emplacement de la balise dans le code HTML. Donc si l'on veut déplacer la position d'une applet dans une page HTML, il suffit de déplacer le code compris entre les deux balises <APPLET CODE =.....> et </APPLET>.

Voici une page HTML dont le titre est "Simple Applet " et ne contenant qu'une applet :

```
<HTML>
  <HEAD>

    <TITLE> Simple Applet </TITLE>

  </HEAD>

  <BODY>
   <APPLET CODE="AppletSimple.class" WIDTH=200 HEIGHT=100>
   </APPLET>
  </BODY>

</HTML>
```

Il y a donc des paramètres obligatoires à transmettre à une applet, ce sont les paramètres CODE, WIDTH et HEIGHT :

Paramètre	signification
CODE	le nomp du fichier contenant la classe applet à afficher.
WIDTH	la largeur de l'applet au démarrage dans la page HTML (en pixels)
HEIGHT	la hauteur de l'applet au démarrage dans la page HTML (en pixels)

A côté de ces paramètres obligatoires existent des paramètres facultatifs relatifs au positionnement et à l'agencement de l'applet à l'intérieur de la page HTML (align, alt, hspace, vspace, codebase, name).

La balise interne PARAM

Il est possible, à l'intérieur du corps de texte entre les balises <APPLET CODE =.....> et

</APPLET> , d'introduire un marqueur interne de paramètre <PARAM .....>, que le fichier HTML transmet à l'applet. Ce marqueur possède obligatoirement deux attributs,soit **name** le nom du paramètre et **value** la valeur du paramètre sous forme d'une chaîne de caractères :

<PARAM name = "NomduParam1" value = "une valeur quelconque">

La classe **applet** dispose d'une méthode permettant de récupérer la valeur associée à un paramètre dont le nom est connu (il doit être strictement le même que celui qui se trouve dans le texte HTML !), cette méthode se dénomme getParameter.

Ci-dessous le même applet que précédemment à qui l'on passe un paramètre de nom **par1** dans le marqueur PARAM.

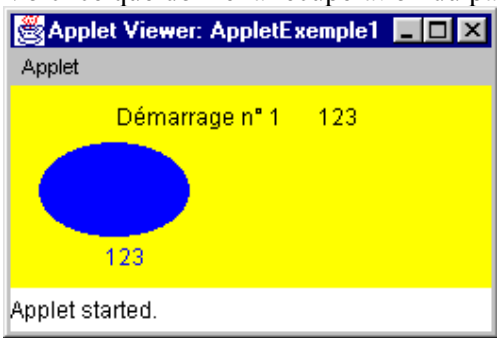
Code Java de l'applet

```
import java.applet.*;
public class AppletExemple1 extends Applet
{   Label etiq = new Label("Démarrage n&deg; ");
    int nbrDemarr = 0 ;
    String valparam ;
    public void init( ) {
    this.setBackground(Color.yellow);
    this.add(etiq);
    valparam = this.getParameter("par1");
    this.add(new Label(valparam));
   }
   public void start( ) {
     nbrDemarr++;
     etiq.setText( "Démarrage n&deg; "+String.valueOf ( nbrDemarr ) );
   }
  public void paint (Graphics x) {
    this.setForeground(Color.blue);
    x.fillOval(15,30,80,50);
    x.drawString(valparam,50,95);
  }
}
```

Code HTML d'affichage de l'applet

```
<HTML>
<HEAD>
<TITLE> Filière CCI </TITLE>
</HEAD>
<BODY>

<APPLET CODE="AppletExemple1" WIDTH=250 HEIGHT=150>
<PARAM NAME = "par1" VALUE = "123">
</APPLET>

</BODY>
</HTML>
```

Voici ce que donne la récupération du paramètre



Nous ne sommes pas limités au passage d'un seul paramètre, il suffit de mettre autant de marqueur PARAM avec des noms de paramètres différents que l'on veut.

Soit l'exemple précédent repris avec deux paramètres : **par1** et **par2**

Code Java de l'applet

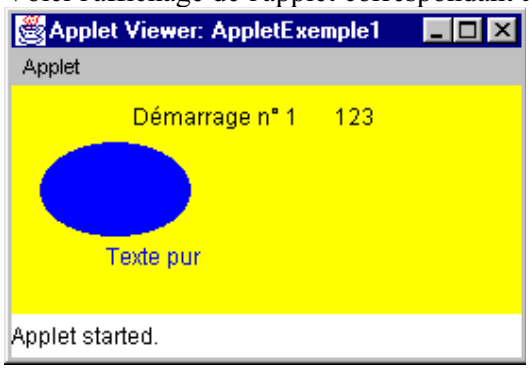
```
import java.applet.*;
public class AppletExemple1 extends Applet
{   Label etiq = new Label("Démarrage n&deg; ");
     int nbrDemarr = 0 ;
     String valparam ;
     public void init( ) {
     this.setBackground(Color.yellow);
     this.add(etiq);
     valparam = this.getParameter("par1");
     this.add(new Label(valparam));
    }
    public void start( ) {
      nbrDemarr++;
      etiq.setText( "Démarrage n&deg; "+String.valueOf ( nbrDemarr ) );
    }
   public void paint (Graphics x) {
     this.setForeground(Color.blue);
     x.fillOval(15,30,80,50);
     x.drawString(getParameter("par2"),50,95);
   }
}
```

La valeur de **par1** reste la même soit "123", celle de **par2** est la phrase "Texte pur"

Code HTML d'affichage de l'applet

```
<HTML>
<HEAD>  <TITLE> Filière CCI </TITLE> </HEAD>
<BODY>

    <APPLET CODE="AppletExemple1" WIDTH=250 HEIGHT=150>
    <PARAM  NAME = "par1"  VALUE = "123">
    <PARAM  NAME = "par2" VALUE = "Texte pur">
    </APPLET>

</BODY>
</HTML>
```

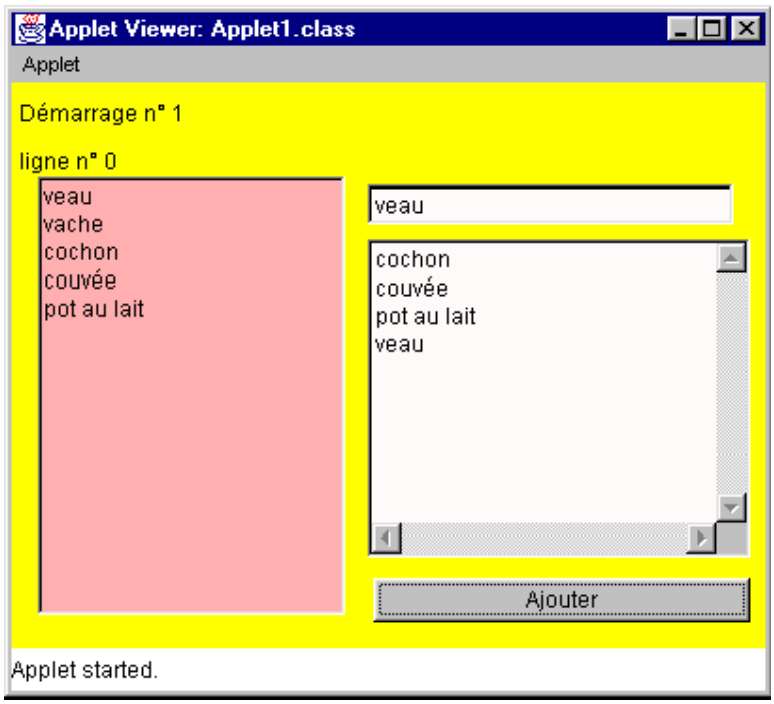
Voici l'affichage de l'applet correspondant à cette combinaison :



AWT dans les applets : exemples

Comme une applet est une classe héritant des Panel, c'est donc un conteneur de composants et donc tout ce que nous avons appris à utiliser dans les classes du package AWT, reste valide pour une applet.

Exemple - 1 : Interface à quatre composants



```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

/* interception du double click de souris par actionPerformed :
- c'est le double click sur un élément de la liste qui déclenche l'action.
list1.addActionListener(new java.awt.event.ActionListener() {

     public void actionPerformed(ActionEvent e) {
       list1_actionPerformed(e);
     }
   });
- même gestion actionPerformed pour le click sur le bouton :
     c'est le click sur le bouton qui déclenche l'action.
*/
 public class Applet0 extends Applet {
       boolean isStandalone = false;
       Label etiq = new Label("Démarrage n&deg;");
       Label numero = new Label("ligne n&deg;");
       int nbrDemarr = 0;
       List list1 = new List();
       TextField textField1 = new TextField();
       TextArea textArea1 = new TextArea();
       Button button1 = new Button();

       // Initialiser l'applet
       public void init( )  {
         this.setSize(new Dimension(400,300));
         this.setLayout(null);
         this.setBackground(Color.yellow);
         etiq.setBounds(new Rectangle(4, 4, 163, 23));
         numero.setBounds(new Rectangle(4, 27, 280, 23));
         list1.setBackground(Color.pink);
         list1.setBounds(new Rectangle(14, 50, 163, 233));
         list1.add("veau");
         list1.add("vache");
         list1.add("cochon");
         list1.add("couvée");
         list1.add("pot au lait");
         list1.addActionListener(new java.awt.event.ActionListener() {
         public void actionPerformed(ActionEvent e) {
               list1_actionPerformed(e);
           }
          });
         button1.addActionListener(new java.awt.event.ActionListener() {
         public void actionPerformed(ActionEvent e) {
               button1_actionPerformed(e);
           }
          });
         textField1.setBackground(Color.pink);
         textField1.setBounds(new Rectangle(189, 54, 194, 21));
         textArea1.setBackground(Color.pink);
         textArea1.setBounds(new Rectangle(189, 83, 203, 169));
         button1.setBounds(new Rectangle(192, 263, 200, 23));
         button1.setLabel("Ajouter");
         this.add(list1, null);
         this.add(etiq, null);
         this.add(numero, null);
```

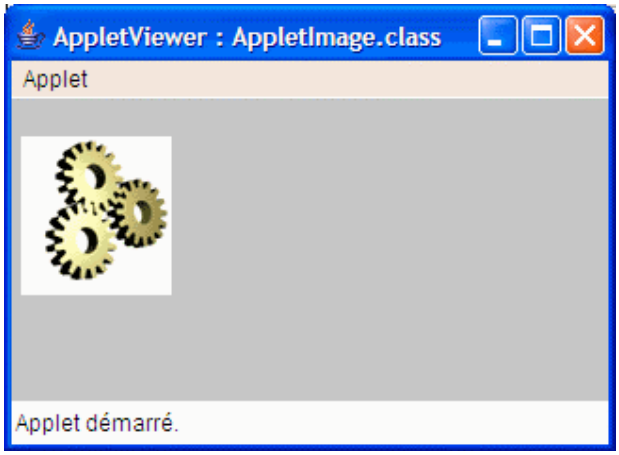
```
        this.add(textField1, null);
        this.add(textArea1, null);
        this.add(button1, null);
        this.setForeground(Color.black);
      }

      // Démarrage de l'applet
     public void start( ) {
       nbrDemarr++;
       etiq.setText("Démarrage n&deg; "+String.valueOf(nbrDemarr));
      }

     // Méthodes d'événement redéfinies
     void list1_actionPerformed(ActionEvent e) {
       int rang = list1.getSelectedIndex( );
       numero.setText("ligne n&deg; "+String.valueOf(rang));
       if (rang>=0) {
         textField1.setText(list1.getSelectedItem( ));
        list1.deselect(list1.getSelectedIndex( ));
        }
       else textField1.setText("rien de sélectionné!");
       }

     void button1_actionPerformed(ActionEvent e) {
         textArea1.append(textField1.getText( )+"\n");
      }
    }
```

On pouvait aussi intercepter les événements au bas niveau directement sur le click de souris, en utilisant toujours des classes anonymes, dérivant cette fois de MouseAdapter et en redéfinissant la méthode mouseClicked :

```
list1.addMouseListener(new MouseAdapter() {
                public void mouseClicked(MouseEvent e) {
                list1_mouseClicked(e);
                }
          });

button1.addMouseListener(new MouseAdapter() {
              public void mouseClicked(MouseEvent e) {
                button1_mouseClicked(e);
                }
        });

void list1_mouseClicked(MouseEvent e) {
// getClickCount indique combien de click ont été efectués sur l'objet (double click = 2 clicks)
     if (e.getClickCount( ) == 2) {
      int rang = list1.getSelectedIndex();
      numero.setText("ligne n&deg; "+String.valueOf(rang));
      if (rang>=0) {
         textField1.setText(list1.getSelectedItem());
       list1.deselect(list1.getSelectedIndex());
        }
      else textField1.setText("rien de sélectionné!");
      }
 }

void button1_mouseClicked(MouseEvent e) {
// remarque : un click = un pressed + un released
```

```
        textArea1.append(textField1.getText()+"\n");
}
```

Exemple - 2 : Afficher une image (gif animé)




```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

   public class AppletImage extends Applet {
     Image uneImage;// une référence d'objet de type Image

      // Initialiser l'applet
      public void init( )  {
        this.setBackground(Color.lightGray); //couleur du fond
        uneImage = getImage(this.getCodeBase( ),"rouage.gif");
        this.setSize(200,160);
      }
      // Dessinement de l'applet
      public void paint(Graphics g ) {
       if(uneImage != null)
       g.drawImage(uneImage,5,20,this);
      }
   }
```

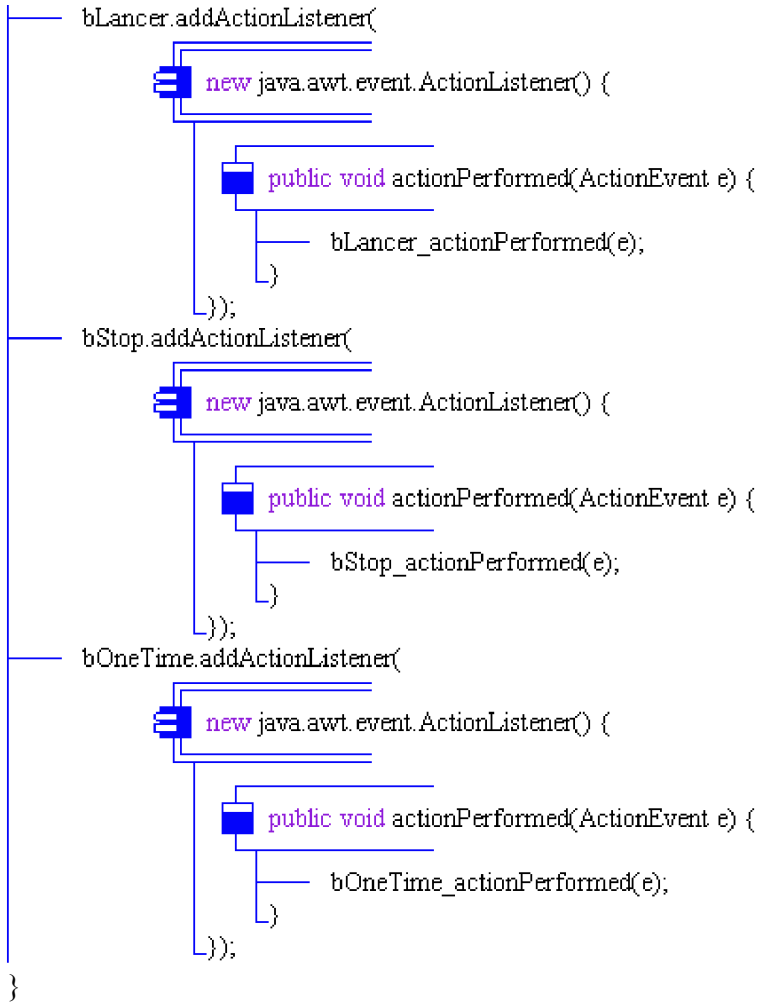
Exemple - 3 : Jouer un morceau de musique

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
```

```
public class AppletMusic1 extends Applet{

    AudioClip music = null;
    Button bLancer = new Button("Ecouter la musique en continu");
    Button bStop = new Button("Stopper la musique");
    Button bOneTime = new Button("Ecouter la musique une seule fois");

    public void init(){
        +

    void bLancer_actionPerformed(ActionEvent e){
        // musique en continu
        music.loop();
    }
    void bStop_actionPerformed(ActionEvent e){
        //Arrêter la musique
        music.stop();
    }
    void bOneTime_actionPerformed(ActionEvent e){
        //musique une seule fois
        music.play();
    }

    public void destroy(){
        music.stop();
    }
}
```

Contenu de la méthode init ( ) :

```
public void init(){
    music = getAudioClip(getCodeBase(),"20TH_CEN.WAV");
    this.setSize(new Dimension(400,100));
    add(bLancer);
    add(bStop);
    add(bOneTime);
```

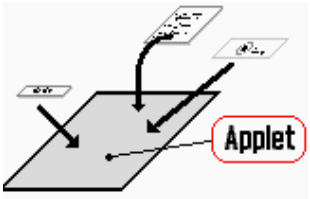

bLancer.addActionListener(
new java.awt.event.ActionListener() {
public void actionPerformed(ActionEvent e) {
bLancer_actionPerformed(e);
}
));
bStop.addActionListener(
new java.awt.event.ActionListener() {
public void actionPerformed(ActionEvent e) {
bStop_actionPerformed(e);
}
));
bOneTime.addActionListener(
new java.awt.event.ActionListener() {
public void actionPerformed(ActionEvent e) {
bOneTime_actionPerformed(e);
}
));
}

Afficher des composants, redessiner une Applet Awt - Swing

Objectif : Comparatif de programmation d'affichage et de redessinement de composants visuels sur une applet Awt et sur une applet Swing. On souhaite redessiner les composants visuels déposés sur une applet lorsqu'ils ont été masqués (par une autre fenêtre) et qu'il nous faut les réafficher.

1. Avec la bibliothèque Awt

En Awt on dépose directement des composants visuels avec la méthode add sur une applet de classe **Applet**. En fait le dessin du composant déposé s'effectue sur l'objet **Canvas** de l'applet (**Canvas** est une classe dérivée de Component destinée à traiter des dessins) :



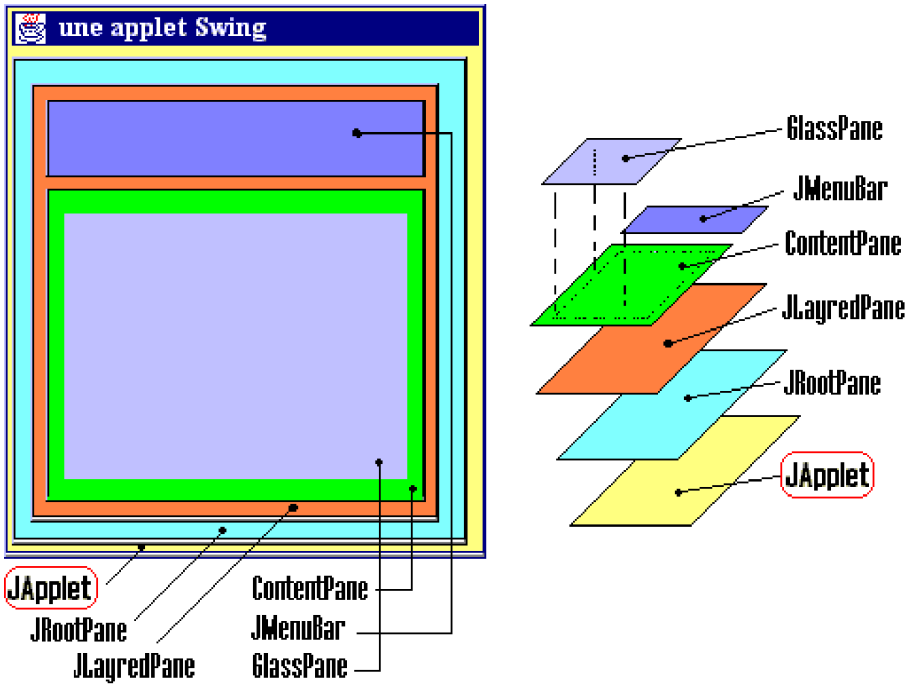
La méthode **public void paint( )** d'une applet Awt est automatiquement appelée par l'applet pour dessiner l'ensemble des graphiques de l'applet. Lorsque l'applet doit être redessinée (la première fois et à chaque fois que l'applet est masquée totalement ou partiellement) c'est la méthode **paint** qui est automatiquement appelée. Nous avons déjà appris que si nous voulions effectuer des dessins spécifiques ou des actions particulières lors du redessinement de l'applet nous devions **redéfinir la méthode paint** de l'applet.

Lorsque l'on dépose un composant sur une applet de classe **Applet**, le parent du composant est l'applet elle-même.

2. Avec la bibliothèque Swing

Le travail est plus complexe avec Swing, en effet le **JApplet** comme le **JFrame,** est un conteneur de composants visuels qui dispose de 4 niveaux de superposition d'objets à qui est déléguée la gestion du contenu du **JApplet**.

2.1 Les 4 couches superposées d'affichage d'une JApplet



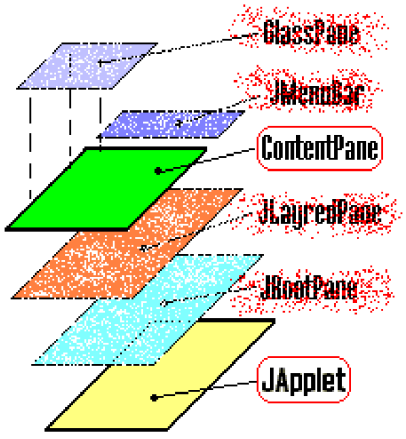
La racine JRootPane, les couches JLayredPane et GlassPane servent essentiellement à l'organisation des menus et du look and feel. Nous rappellons que seule la couche ContentPane doit être utilisée par le développeur pour écrire ses programmes.

Tout ce que l'on dépose sur une **JApplet** doit en fait l'être sur son **ContentPane** qui est une instance de la classe **Container**. C'est pourquoi dans les exemples qui sont proposés vous trouvez l'instruction :

```
this.getContentPane( ).add(....)
```

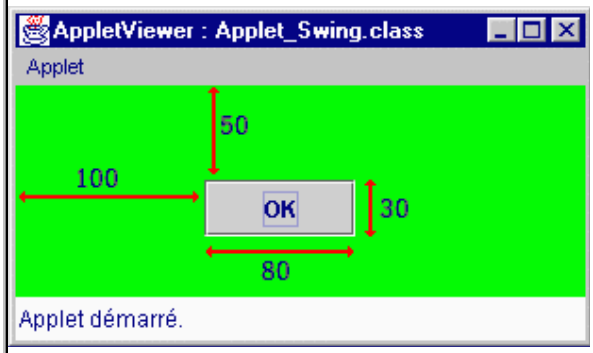
2.2 Les deux seules couches utilisées

Parmi ces couches nous n'en utiliserons que deux :



Colorons l'applet en jaune, son contentPane en vert et déposons un JButton "OK" sur l'applet (sur son contentPane) :

```
public class Applet_Swing extends JApplet
{
 Container contentPane = getContentPane();
 void Ecrire ( )
 {
  System.out.println("couleur this =" + this.getBackground( ).toString( ));
  System.out.println("couleur contentPane =" + contentPane.getBackground( ).toString( ));
  System.out.println("couleur parent du contentPane =" +
                          contentPane.getParent( ).getBackground( ).toString( ));

 }
 public void init( )
 {
   JButton bouton = new JButton("OK");
   bouton.setBounds(100,50,80,30);
   contentPane.setLayout(null);
   contentPane.add(bouton);
   contentPane.setBackground(Color.green);
   this.setBackground(Color.yellow);
   Ecrire( );
 }
}
/* les valeurs des couleurs en RGB :
yellow = 255,255,0 <--- JApplet
green = 0,255,0   <--- contentPane de JApplet

*/
```

Résultat visuel de l'applet :	Résultat des écritures de l'applet :
	couleur this = java.awt.Color[r=255,g=255,b=0] **yellow** couleur contentPane = java.awt.Color[r=0,g=255,b=0] **green** couleur parent du contentPane = java.awt.Color[r=255,g=255,b=0] **yellow**

Nous remarquons que nous voyons le bouton déposé et positionné sur le contentPane et le fond vert du contentPane, mais pas le fond de l'applet (**this**.setBackground(Color.yellow); représente la coloration du fond de l'applet elle-même).

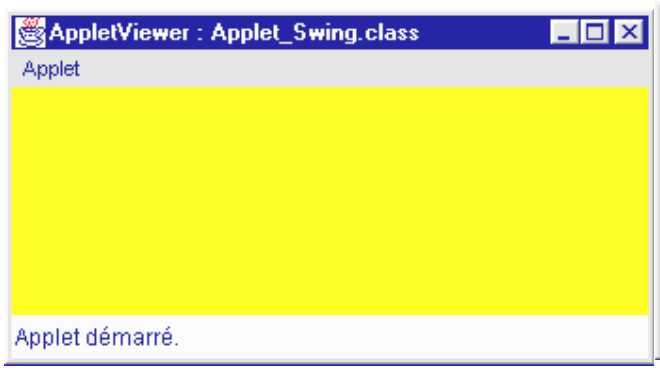
2.3 la méthode paint de l'applet redessine l'applet seulement

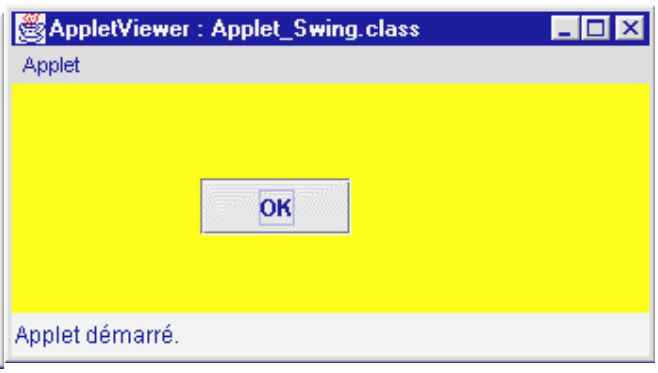
Essayons d'agir comme avec une applet Awt en redéfinissant la méthode paint (**JApplet** dérivant de **Applet** possède la méthode paint) sans mettre de code dans le corps de cette méthode, afin de

voir quel travail effectue la méthode paint :

```
public class Applet_Swing extends JApplet
{
 Container contentPane = getContentPane();
 void Ecrire ( )
 {
  System.out.println("couleur this =" + this.getBackground( ).toString( ));
  System.out.println("couleur contentPane =" + contentPane.getBackground( ).toString( ));
  System.out.println("couleur parent du contentPane =" +
                contentPane.getParent( ).getBackground( ).toString( ));

 }
 public void init( )
 {
   JButton bouton = new JButton("OK");
   bouton.setBounds(100,50,80,30);
   contentPane.setLayout(null);
   contentPane.add(bouton);
   contentPane.setBackground(Color.green);
   this.setBackground(Color.yellow);
   Ecrire( );
 }

 public void  paint (Graphics g)
 { // redéfinition de la méthode : pas d'action nouvelle
 }

}
/* les valeurs des couleurs en RGB :
yellow = 255,255,0 <--- JApplet
green = 0,255,0   <--- contentPane de JApplet

*/
```

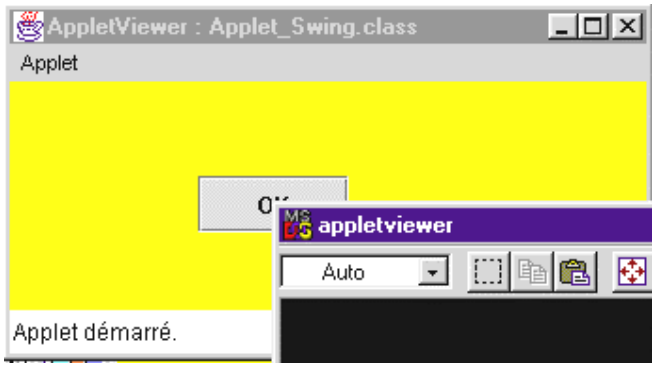
Résultats obtenus lors de l'exécution de l'applet :



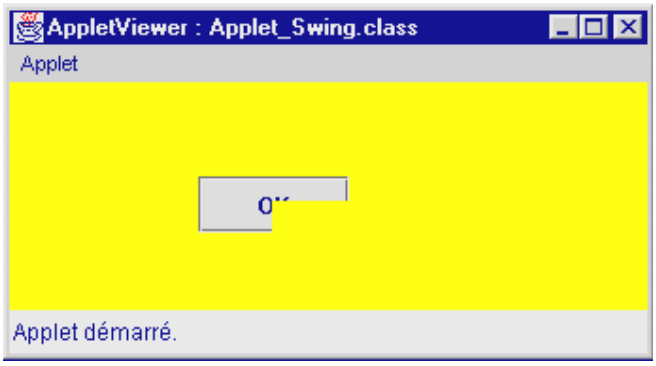


Nous nous rendons compte que cette fois-ci, le fond du contentPane vert n'a pas été affiché, mais que c'est le fond jaune de l'applet elle-même qui l'est. Le bouton OK n'est pas visible bien qu'il existe sur l'applet.

Nous masquons entièrement l'applet par une fiche, puis nous le démasquons : le bouton OK est alors redessiné par l'applet mais pas par la méthode **paint**. Si plutot nous faisons passer la souris dans la zone supposée du bouton OK c'est le bouton qui se redessine lui-même mais pas la méthode **paint**



En effet après avoir fait apparaître le bouton masquons partiellement l'applet avec une autre fenêtre, puis démasquons l'applet en rendant la fenêtre de l'applet active.



Nous voyons que l'applet a bien été rafraîchie mais que l'image du bouton ne l'a pas été, donc la méthode **paint** redéfinie redessine l'applet, mais n'a pas redessiné le bouton OK qui est déposé sur le contentPane.

Conclusion n°1
La méthode **paint** redéfinie redessine l'applet, mais ne redessine pas les objets du contentPane.

2.4 Variations : appel à la méthode paint de la super classe

1°) Sachant qu'une applet de classe **Applet** de Awt se redessine avec tous ses composants :

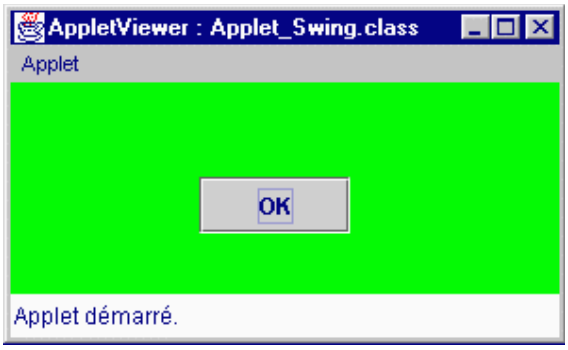
```
java.awt.Container
        |
        +--java.awt.Panel
            |
            +--java.applet.Applet
```

2°) Sachant que **JApplet** de Swing est une classe fille de **Applet** , nous nous proposons alors de faire appel à la méthode **paint** de l'ancêtre (la super classe) qui est la classe **Applet** de Awt. Dans ce cas nous forcerons l'applet Swing à réagir comme une applet Awt :

```
java.applet.Applet
                |
                +--javax.swing.JApplet
```

```
public void  paint (Graphics g)
{ // redéfinition de la méthode : on appelle la méthode de l'ancêtre
    super.paint ( g );
}
```

Nouveau résultat d'exécution :



Nous remarquons que l'appel à la méthode **paint** de la super classe provoque un affichage correct, le masquage et le démasquage fonctionnent correctement aussi. C'est cette solution que nous avons adoptée dans les exercices corrigés.

Problème potentiel d'instabilité de Java

> Les concepteurs de Swing déconseillent cette démarche car la surcharge de la méthode **paint** de l'applet (en fait de la classe **Container**) pourrait interférer dans certains cas avec la méthode **paint** du **JComponent** qui surcharge elle-même déjà la méthode **paint** de la classe **Container**.

3. Méthode paintComponent pour utiliser le redessinement

3.1 Généralités pour un composant

Il est conseillé en général d'utiliser systématiquement la méthode **paintComponent** de chaque composant qui est appelée automatiquement dès que l'applet est redessinée. En fait chaque composant reçoit une notification de redessinement et la méthode **paintComponent** de chaque composant est exécutée. Si en outre nous voulons effectuer des actions spécifiques pendant le redessinement d'un composant, nous devons alors surcharger la méthode **paintComponent** de ce composant.

3.2 Mettez un JPanel dans votre applet

Le composant de base qui encapsule tous les autres composants d'une applet **JApplet** est son **contentPane**. Etant donné que contentPane est un objet nous ne pouvons donc pas surcharger la méthode du **contentPane** (on peut surcharger une méthode dans une classe) nous créons une nouvelle classe héritant des **JPanel** :

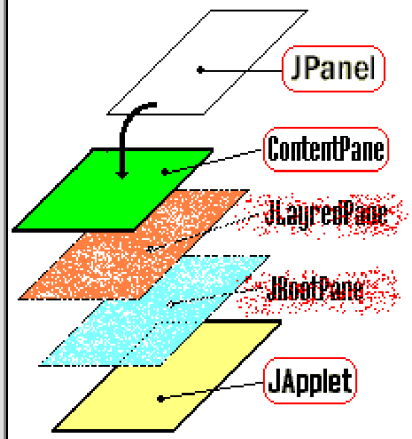
```
class JPanelApplet extends JPanel  {    //-- constructeur :
    JPanelApplet( )       {
      setBackground(Color.white); // le fond est blanc
    }
}
```

Nous allons rajouter une couche supplémentaire sur l'applet avec un nouvel objet instancié à partir de cette nouvelle classe héritant des **JPanel** dont nous surchargerons la méthode **paintComponent** (puisqu'elle est invoquée automatiquement par l'applet). Nous déposerons ce **JPanel** sur le **contentPane** de l'applet.

```
class JPanelApplet extends JPanel  {
   JPanelApplet( ) // constructeur
   {
        setBackground(Color.white); // le fond est blanc
    }
   public void paintComponent (Graphics g)
   { // le redessinement est traité ici
     super.paintComponent(g);
     g.drawString("Information dessinée sur le fond graphique", 10, 40);
    }
}
```

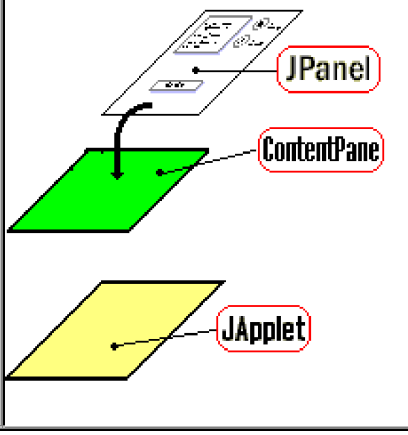
Nous instancions un objet panneau de cette classe et l'ajoutons au **contentPane** de l'applet :

```
public class Applet_AfficheSwing extends JApplet {
  JPanelApplet panneau;

  public void init()
  {
   Container contentPane = getContentPane( );
   panneau = new JPanelApplet( );
   contentPane.add(panneau);
   contentPane.setBackground(Color.green);
   this.setBackground(Color.yellow);
  }
}
```

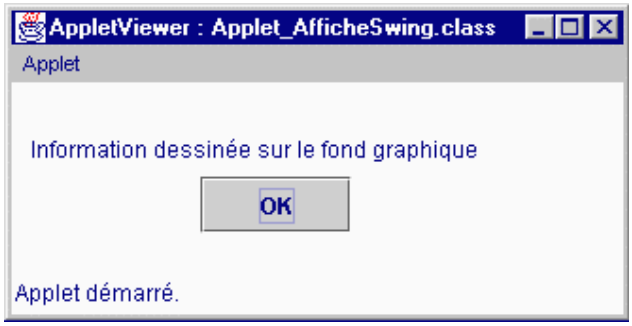


Nous déposerons tous nos composants sur ce **JPanel**, ceci permettra aussi de transformer plus facilement des applications encapsulées dans un **JPanel** qui peut être identiquement déposé sur un **JFrame** vers une applet Swing..

```
public class Applet_AfficheSwing extends JApplet {
  JPanelApplet panneau;

  public void init( )  {
   Container contentPane = getContentPane( );
   panneau = new JPanelApplet( );
   contentPane.add(panneau);
   contentPane.setBackground(Color.green);
   this.setBackground(Color.yellow);
   JButton bouton = new JButton("OK");
   bouton.setBounds(100,50,80,30);
   panneau.setLayout(null);
   panneau.add(bouton);
  }
}
```



Voici le résultat de l'exécution de l'applet ainsi construite:



Conclusion

Toute action à effectuer lors du redessinement de l'applet pourra donc être intégrée dans la méthode surchargée **paintComponent** de la classe **JPanelApplet**, le code minimal d'une applet Swing ressemblera alors à ceci :

```
class JPanelApplet extends JPanel
 {
   JPanelApplet( ) // constructeur
   {
        .....
   }

  public void paintComponent (Graphics g)
   { // le redessinement est traité ici
     super.paintComponent(g);
     //..... vos actions spécifiques lors du redessinement
   }
}

public class Applet_AfficheSwing extends JApplet
{
  JPanelApplet panneau;

  public void init( )
  {
   Container contentPane = getContentPane( );
   panneau = new JPanelApplet( );
   contentPane.add(panneau);
   ..... // les composants sont ajoutés à l'objet "panneau"
  }
}
```

Classes internes ,exception , threads

Le contenu de ce thème :

Les classes internes

Les exceptions

Le multi-threading

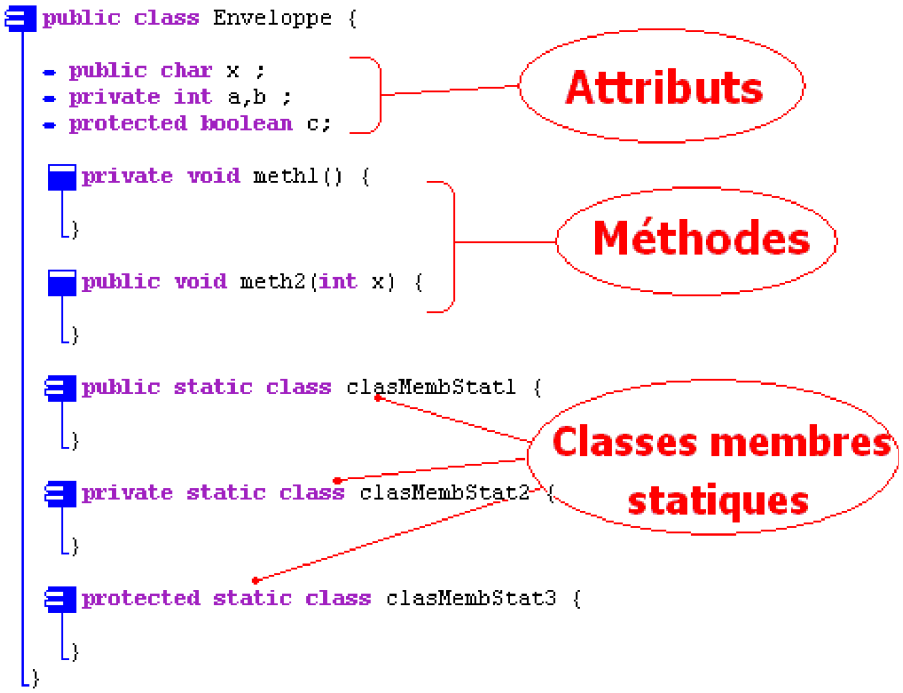
Les classes internes

Java2

Depuis la version Java 2, le langage java possède quatre variétés supplémentaires de classes, que nous regroupons sous le vocable général de **classes internes**.

Les classes membres statiques

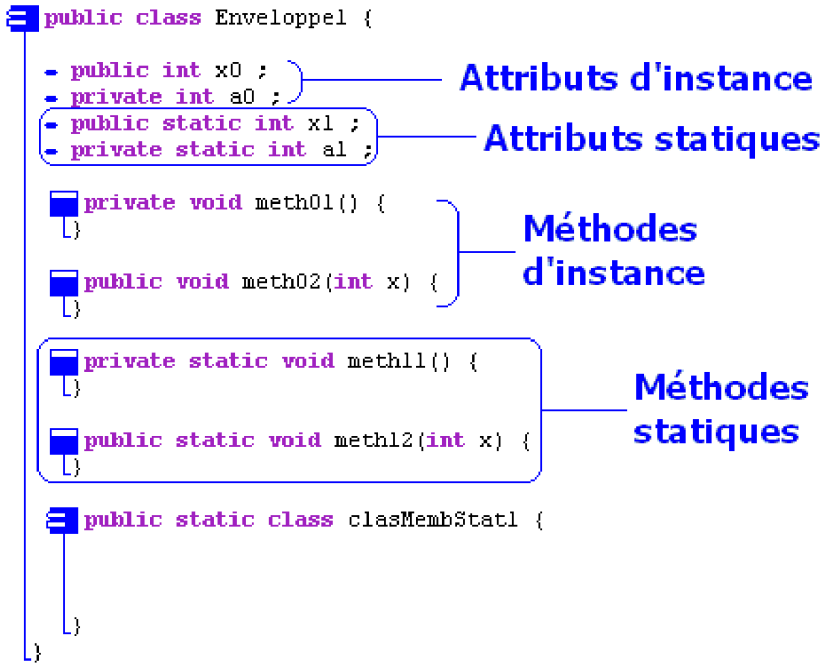
Exemple de classes membres statiques



Définition des classes membres statiques

Une classe membre statique est une classe java définie dans la partie déclaration des membres d'une autre classe qui la contient (nommée classe englobante), puis qualifiée par le modificateur **static**. ❑ Une classe membre statique est instanciable. ❑ Une classe membre statique ne peut pas être associée à un objet instancié de la classe englobante .	***Syntaxe*** : **public class** Enveloppe **{** < membres genre attributs > < membres genre méthodes > < membres genre classes membres statiques > **}**

Une classe membre statique est analogue aux autres membres statiques de la classe dans laquelle elle est déclarée (notée Enveloppe ici) :



On peut dans une autre classe, instancier un objet de classe statique, à condition d'utiliser le qualificateur de visibilité qu'est le nom de la classe englobante afin d'accéder à la classe interne. Une classe membre statique se comporte en fait comme une classe ordinaire relativement à l'instanciation :

Soit par exemple, la classe **Autre** souhaite déclarer et instancier un objet Obj0 de classe public clasMembStat1 :

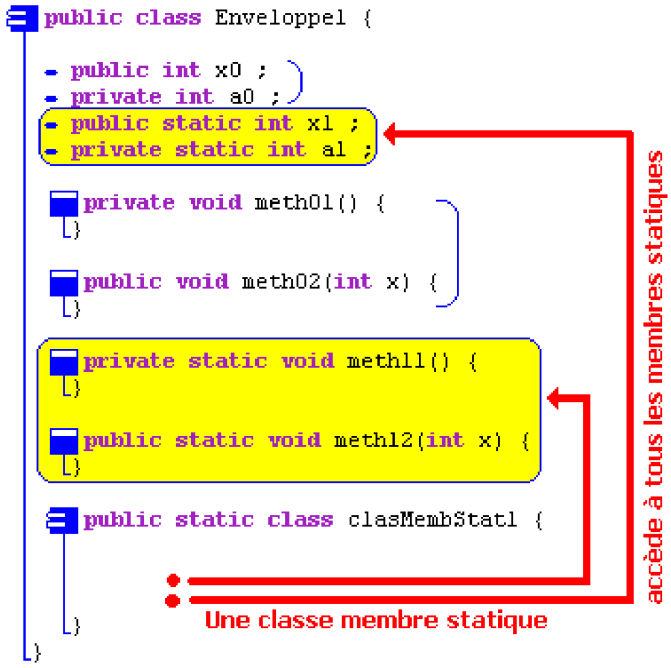
```
class Autre {
        Enveloppe1.clasMembStat1 Obj0 = new Enveloppe1.clasMembStat1( ) ;
        .......
}
```

Soit en plus à instancier un objet local Obj1 dans une méthode de la classe **Autre** :

```
class Autre{
     Enveloppe1.clasMembStat1 Obj0 = new Enveloppe1.clasMembStat1() ;
      void meth( ){
        Enveloppe1.clasMembStat1 Obj1 = new Enveloppe1.clasMembStat1() ;
     }
}
```

Caractérisation d'une classe membre statique

Une classe membre statique accède à **tous les membres statiques** de sa classe englobante qu'ils soient **publics** ou **privés**, sans nécessiter d'utiliser le nom de la classe englobante pour accéder aux membres (raccourcis d'écriture) :



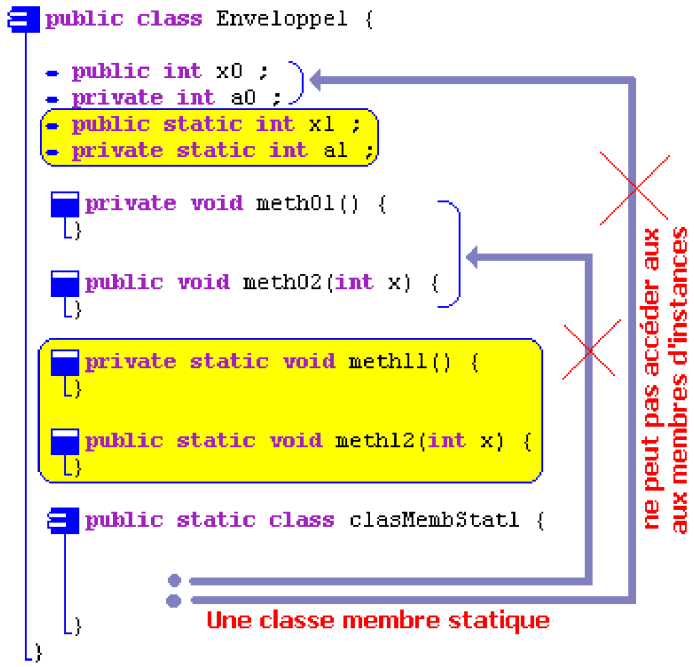
Exemple :
Ci-dessous la classe membre statique clasMembStat1 contient une méthode "**public void** meth( )", qui invoque la méthode de classe(statique) **void** meth12 de la classe englobante Enveloppe1, en lui passant un paramètre effectif statique **int** a1. Nous écrivons 4 appels de la méthode meth12( ) :

```
public static class clasMembStat1 {

        public void meth( ){
         meth12(a1);
          Enveloppe1.meth12(a1);
          meth12(Enveloppe1.a1);
         Enveloppe1.meth12(Enveloppe1.a1);
       }
}
```

Nous notons que les 4 appels de méthodes sont strictement équivalents, en effet ils différent uniquement par l'utilisation ou la non utilisation de raccourcis d'écriture.

- Le quatrième appel : "Enveloppe1.meth12(Enveloppe1.a1)" est celui qui utilise le nom de la classe englobante pour accéder aux membres statiques comme le prévoit la syntaxe générale.
- Le premier appel : "meth12(a1)" est celui qui utilise la syntaxe particulière aux classes membres statiques qui autorise des raccourcis d'écriture.

Une classe membre statique n'a pas accès aux membres d'instance de la classe englobante :



Remarque importante :

Un objet de **classe interne statique** ne comporte pas de référence à l'objet de classe externe qui l'a créé.

Remarque Interface statique :

Une interface peut être déclarée en interface interne statique comme une classe membre statique en utilisant comme pour une classe membre statique le mot clef **static**.

Syntaxe :

public class Enveloppe **{**
< membres genre attributs >
< membres genre méthodes >
< membres genre classes membres statiques >
< membres genre interfaces statiques >
}

Exemple **:**

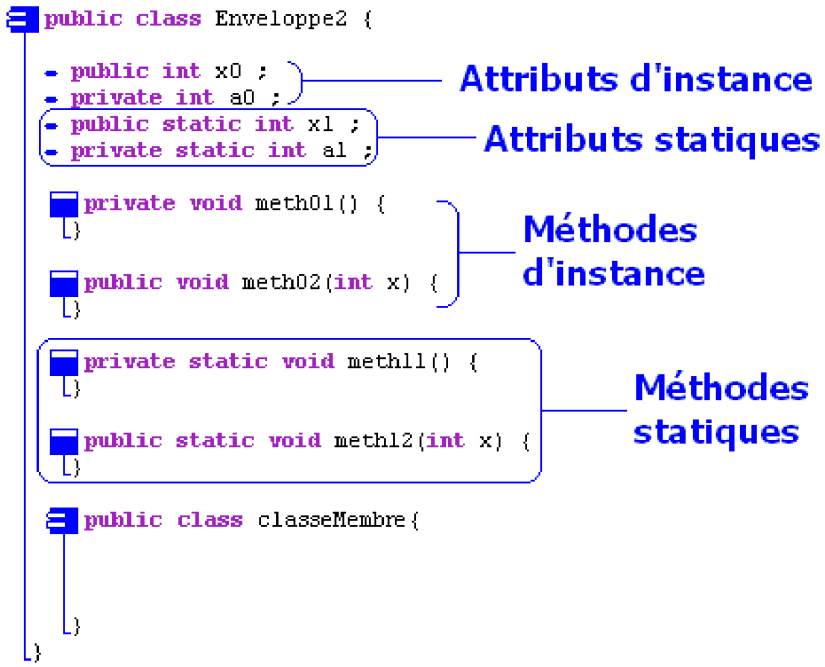
public class Enveloppe **{**
public static Interface Interfstat {
.....
}
}

Les classes membres

Définition des classes membres

Une classe membre est une classe java définie dans la partie déclaration des membres d'une autre classe qui la contient (nommée classe englobante). ❑ Une classe membre est instanciable. ❑ Une classe membre est associée à un objet instancié de la classe englobante .	***Syntaxe*** : **public class** Enveloppe { < membres genre attributs > < membres genre méthodes > < membres genre classes membres statiques > < membres genre classes membres > }

Une classe membre se déclare **d'**une manière identique **à une classe membre statique** et aux autres membres de la classe englobante (notée Enveloppe2 ici) :



On peut dans une autre classe, instancier un objet de classe membre, à condition d'utiliser le qualificateur de visibilité qu'est le nom de la classe englobante afin d'accéder à la classe interne. Une classe membre se comporte en fait comme une classe ordinaire relativement à l'instanciation :

Soit par exemple, la classe **Autre** souhaite déclarer et instancier un objet Obj0 de classe public classeMembre :

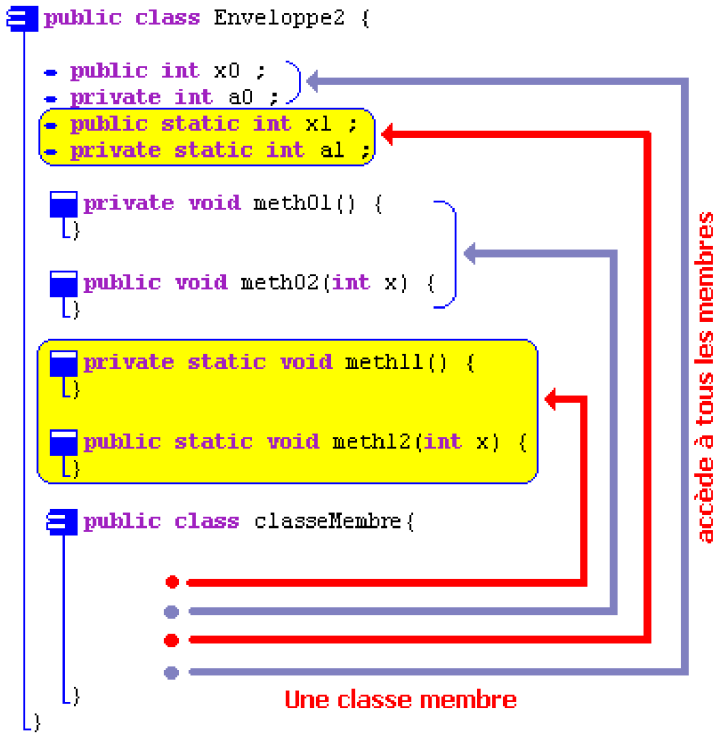
```
class Autre {
        Enveloppe1.classeMembre Obj0 = new Enveloppe1.classeMembre( ) ;
        .....
```

```
}
```

Soit en plus à instancier un objet local Obj1 dans une méthode de la classe **Autre** :

```
class Autre{
     Enveloppe1.classeMembre Obj0 = new Enveloppe1.classeMembre() ;
      void meth( ){
        Enveloppe1.classeMembre Obj1 = new Enveloppe1.classeMembre() ;
      }
}
```

Caractérisation d'une classe membre



Remarque importante :

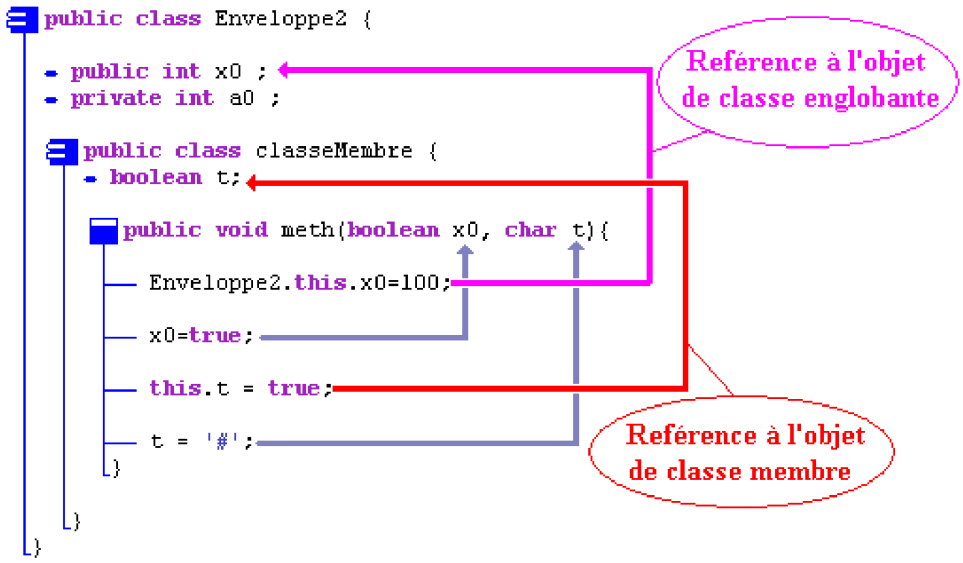
Les classes membres ont les mêmes fonctionalités syntaxiques que les classes membres statiques. La différence notable entre ces deux genres de classes se situe dans l'accès à l'instance d'objet de classe englobante : Un objet de **classe interne non statique comporte une référence à l'objet de classe externe** qui l'a créé, alors qu'il n'en est rien pour une classe membre statique. L'accès a lieu à travers le mot clef **this** qualifié par le nom de la classe englobante : "ClasseEnveloppe.**this**" .

Exemple d'accès à la référence de l'objet externe :

Question : comment, dans la classe interne membre classeMembre, ci-dessous accéder aux différents champs masqués x0 et t :

```
public class Enveloppe2 {

    public int x0 ;
    private int a0 ;
     ........
     public class classeMembre {
      boolean t ;
      void meth( boolean x0, char t ) {
       /*
          accéder au membre x0 de Enveloppe2 ;
          accéder au  paramètre formel x0 de meth ;
          accéder au membre t de classeMembre ;
          accéder au  paramètre formel t de meth ;
          */
      }
     ........
    }
  }
```

Réponse : en utilisant le mot clef **this** à la fois pour obtenir la référence de l'objet de classe classeMembre et Enveloppe2.**this** pour obtenir la reference sur l'objet externe de classe englobante :



Remarque Interface non déclarée static :

Une interface ne peut pas être déclarée en interface interne non statique comme une classe membre ordinaire car on ne peut pas instancier d'objetà partir d'une interface. Si vous oubliez le mot clef **static**, le compilateur java le "rajoutera" automatiquement et l'interface sera considérée comme statique !

Les classes locales

Définition des classes locales

Une classe locale est déclarée au sein d'un **bloc de code** Java.généralement dans le corps d'une méthode d'une autre classe qui la contient (nommée classe englobante). Elle adopte un schéma de visibilité gross-mode semblable à celui d'une variable locale. ❑ Une classe locale est instanciable comme une classe membre. ❑ Une classe locale est associée à un objet instancié de la classe englobante.	***Syntaxe*** : **public class** Enveloppe **{** < membres genre attributs > < membres genre méthodes > < membres genre classes membres statiques > < membres genre classes membres > < membres genre interfaces statiques > < membres genre classes locales > **}**

Une classe locale peut être déclarée au sein d'un quelconque bloc de code Java. Ci-dessous 3 classes locales déclarées chacune dans 3 blocs de code différents :

```
public class Enveloppe3 {

  public int x0 ;
  private int a0 ;

  public void meth(char x0 ) {

    class classeLocale1 {
      //......
    }

    for(int i=0;i<10;i++) {
      class classeLocale2 {
        //......
      }

      classeLocale2 Obj2;

      if (i==5) {

        Obj2 = new classeLocale2();

        class classeLocale3 {
          //......
        }
      }
    }
  }
}
```

Caractérisation d'une classe locale

Une classe locale n'est visible et utilisable qu'au sein d'un **bloc de code** Java.où

elle a été déclarée.

Une classe locale peut utiliser des variables (locales, d'instances ou paramètres) visibles dans le bloc ou elle est déclarée à la condition impérative que ces variables aient été déclarées en mode **final**.

Une classe locale peut utiliser des variables d'instances comme une classe membre.

A la lumière de la définition précédente corrigeons le code source de la classe ci-dessous, dans laquelle le compilateur Java signale 3 erreurs :

```
public class Enveloppe3 {
    public int x0 = 5 ;
    private int a0 ;

     public void meth(char x0 ) {
       int x1=100;
       class classeLocale1 {
         int j = x1; // le int x1 local à meth( )
        int c =  x0;  // le int x0 de Enveloppe3
       }
       for ( int i = 0;  i<10; i++) {
        int k = i;
        class classeLocale2 {
          int j = k + x1 + x0; // le k local au for
        }
       }
    }
  }
```

Après compilation, le compilateur java signale des accès érronés à des variables non final :

C:\j2sdk1.4.2\bin\javac - 3 errors :

local variable x1 is accessed from within inner class; needs to be declared final
int j = x1;

local variable x0 is accessed from within inner class; needs to be declared final
int c = x0;

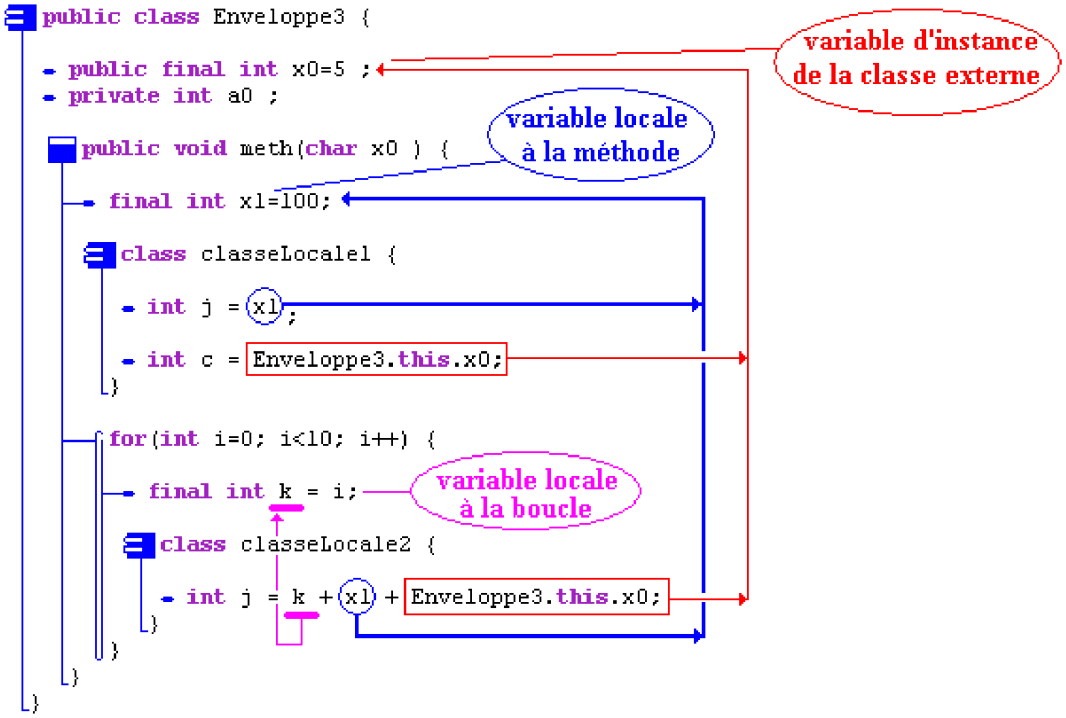
local variable k is accessed from within inner class; needs to be declared final
int j = k + x1 + x0;

On propose une première correction :

```
public class Enveloppe3 {
    public int x0=5 ;
    private int a0 ;

     public void meth ( final char x0 ) {
       final int x1=100;
        class classeLocale1 {
         int j = x1;
         int c =  x0;
```

```
      }
      for ( int i = 0;  i<10; i++) {
        final int k = i;
         class classeLocale2 {
           int j = k + x1 + x0;
         }
      }
    }
  }
```

Cette fois-ci le compilateur accepte le code source comme correct. Toutefois cette solution ne correspond pas à la définition de l'instruction "**int** j = k + x1 + x0; " dans laquelle la variable x0 doit être la variable d'instance de la classe externe Enveloppe3. Ici c'est le paramètre formel "**final char** x0" qui est utilisé. Si l'on veut accéder à la variable d'instance, il faut la mettre en mode final et qualifier l'identificateur x0 par le nom de la future instance de classe Enveloppe3, soit "Enveloppe3.**this**" :



Remarque :

Une classe locale **ne peut pas** être qualifiée en **public**, **private**, **protected** ou **static**.

Les classes anonymes

Définition des classes anonymes

Une classe **anonyme** est une **classe locale qui ne porte pas de nom**.

Une classe anonyme possède toutesles propriétés d'une classe locale.

Comme une classe locale n'a pas de nom vous ne pouvez pas définir un constructeur.

Syntaxe :

```
public class Enveloppe {
   < membres genre attributs >
   < membres genre méthodes >
   < membres genre classes membres statiques >
   < membres genre classes membres >
   < membres genre interfaces statiques >
   < membres genre classes locales >
   < membres genre classes anonymes >
}
```

Une classe anonyme est instanciée immédiatement dans sa déclaration selon une syntaxe spécifique :

new <identif de classe> **(** <liste de paramètes de constructions>**) {** <corps de la classe>**}**

Soit l'exemple ci-dessous , comportant 3 classes Anonyme, Enveloppe4, Utilise :

```
class Anonyme{
  public int a;

  Anonyme(int x){
    a=x*10;
    methA(x);
    methB();
  }

  public void methA(int x){
    a+=2*x;
  }

  public void methB(){}
}

public class Enveloppe4 {

  public void meth(Anonyme x){
  }
}
```

```
class Utilise{
  Enveloppe4 Obj = new Enveloppe4() ;

  void meth(){

    Obj.meth(
        new Anonyme(8){         });

    Obj.meth(
        new Anonyme(12){
          public void methA(int x){
            super.methA(x);
            a -=10;
          }
          public void methB(int x){
            System.out.println("a = "+a);
          }
        });

    Obj.meth(
        new Anonyme(-54){
          public void methB(int x){
            System.out.println("a = "+a);
          }
        });
  }
}
```

La classe Enveloppe4 déclare une méthode public possédant un paramètre formel de type Anonyme :

```
public class Enveloppe4 {

    public void meth(Anonyme x){
    }
}
```

La classe Anonyme déclare un champ entier public a, un constructeur et deux méthodes public :

```
class Anonyme {
     public int a;
   // constructeur:
      Anonyme (int x){
        a = x*10;
        methA(x);
        methB();
      }
      public void methA(int x){
        a += 2*x ;    }

      public void methB( ){ }
  }
```

La classe Utilise déclare un champ objet de type Enveloppe4 et une méthode qui permet la création de 3 classes internes anonymes dérivant chacune de la classe Anonyme :

```
class Utilise{
     Enveloppe4 Obj = new Enveloppe4() ;

      void meth(){
       /* création d'une première instance anonyme de classe dérivant de Anonyme
          avec comme valeur de construction 8.
       */
        Obj.meth( new Anonyme (8){ });

       /* création d'une seconde instance anonyme de classe dérivant de Anonyme
          avec comme valeur de construction 12, redéfinition de la méthode methA et
          redéfinition de la méthodee methB :
       */
        Obj.meth(new Anonyme(12){
                public void methA(int x){
                  super.methA(x);
                   a -= 10;
                }
                public void methB(int x){
                   System.out.println("a = "+a);
                }
              });
       /* création d'une troisième instance anonyme de classe dérivant de Anonyme
          avec comme valeur de construction -54 et redéfinition de la seule méthode methA :
       */
        Obj.meth(new Anonyme(-54){
                public void methB(int x){
                   System.out.println("a = "+a);
                }
              });
    }
  }
```

Caractérisation d'une classe anonyme

- Une classe anonyme étend concrètement une classe déjà existante abstraite ou non, ou bien implémente concrètement une interface.
- Une classe anonyme sert lorsque l'on a besoin d'une classe pour un seul usage unique, elle est définie et instanciée là où elle doit être utilisée.
- L'exemple le plus marquant d'utilisation de classe anonyme est l'instanciation d'un écouteur.

Voyons la différence d'écriture entre la version précédente de la classe Utilise avec 3 objets de classes anonymes et la réécriture avec des classes ordinaires :

On doit d'abord créer 3 classes dérivant de Anonyme :

```
class Anonyme1 extends Anonyme {
    Anonyme1(int x){
       super(x);
    }
}

class Anonyme2 extends Anonyme {
    Anonyme2(int x){
       super(x);
    }
    public void methA(int x){
       super.methA(x);
       a -= 10;
    }
    public void methB(int x){
       System.out.println("a = "+a);
    }
}

class Anonyme3 extends Anonyme {
    Anonyme3(int x){
       super(x);
    }
    public void methB(int x){
       System.out.println("a = "+a);
    }
}
```

Puis enfin définir la classe Utilise, par exemple comme suit :

```
class Utilise{
    Enveloppe4 Obj = new Enveloppe4() ;

void meth() {
      Anonyme1 Obj1 = new Anonyme1(8);
      Anonyme2 Obj2 = new Anonyme2(12);
      Anonyme3 Obj3 = new Anonyme3(-54);
      Obj.meth( Obj1);
      Obj.meth( Obj2 );
      Obj.meth( Obj3 );
   }
  }
```

Vous pourrez comparer l'élégance et la briéveté du code utilisant les classes anonymes par rapport au code classique :

```
class Utilise{
    Enveloppe4 Obj = new Enveloppe4() ;

     void meth(){
      Obj.meth( new Anonyme(8){ });
      Obj.meth(new Anonyme(12){
            public void methA(int x){
              super.methA(x);
               a -= 10;
            }
            public void methB(int x){
               System.out.println("a = "+a);
            }
          });
      Obj.meth(new Anonyme(-54){
            public void methB(int x){
               System.out.println("a = "+a);
            }
          });
   }
  }
```

Les exceptions

Java2

Les exceptions : syntaxe, rôle, classes

Rappelons au lecteur que la sécurité d'une application peut être rendue instable par toute une série de facteurs :

Des problèmes liés au matériel : par exemple la perte subite d'une connection à un port, un disque défectueux... Des actions imprévues de l'utilisateur, entrainant par exemple une division par zéro... Des débordements de stockage dans les structures de données...

Toutefois les faiblesses dans un logiciel pendant son exécution, peuvent survenir : lors des entrées-sorties, lors de calculs mathématiques interdits (comme la division par zéro), lors de fausses manoeuvres de la part de l'utilisateur, ou encore lorsque la connexion à un périphérique est inopinément interrompue, lors d'actions sur les données. Le logiciel doit donc se " ***défendre*** " contre de tels incidents potentiels, nous nommerons cette démarche la programmation défensive !

Progammation défensive
La ***programmation défensive*** est une attitude de pensée consistant à prévoir que le logiciel sera soumis à des défaillances dues à certains paramètres externes ou internes et donc à prévoir une réponse adaptée à chaque type de situation.

En programmation défensive il est possible de protéger directement le code à l'aide de la notion d'exception. L'objectif principal est d'améliorer la qualité de " ***robustesse*** " (définie par B.Meyer) d'un logiciel. L'utilisation des exceptions avec leur mécanisme intégré, autorise la construction rapide et efficace de logiciels robustes.

Rôle d'une exception
Une exception est chargée de signaler un comportement ***exceptionnel*** (mais prévu) d'une partie spécifique d'un logiciel. Dans les langages de programmation actuels, les exceptions font partie du langage lui-même.C'est le cas de Java qui intègre les exceptions comme une classe particulière: la classe **Exception**.Cette classe contient un nombre important de classes dérivées.

Comment agit une exception
Dès qu'une erreur se produit comme un manque de mémoire , un calcul impossible, un fichier inexistant, un transtypage non valide,..., un objet de la classe adéquate dérivée de la classe **Exception** est instancié. Nous dirons que le logiciel " ***déclenche une exception*** ".

Comment gérer une exception dans un programme

Programme sans gestion de l'exception

Soit un programme Java contenant un incident d'exécution (une division par zéro dans l'instruction x = 1/0;) dans la méthode meth() de la classe Action1, cette méthode est appelée dans la classe UseAction1 à travers un objet de classe Action1 :

```
class Action1 {
    public void meth(){
        int x;
        System.out.println("  ...Avant incident");
        x=1/0;
        System.out.println("  ...Après incident");
    }
}

class UseAction1{
    public static void main(String[] Args) {
        Action1 Obj = new Action1();
        System.out.println("Début du programme.");
        Obj.meth();
        System.out.println("Fin du programme.");
    }
}
```

Lors de l'exécution, après avoir affiché les chaînes "**Début du programme**" et " **...Avant incident**", le programme s'arrête et la java machine signale une erreur. Voici ci-dessous l'affichage obtenu sur la console lors de l'exécution :

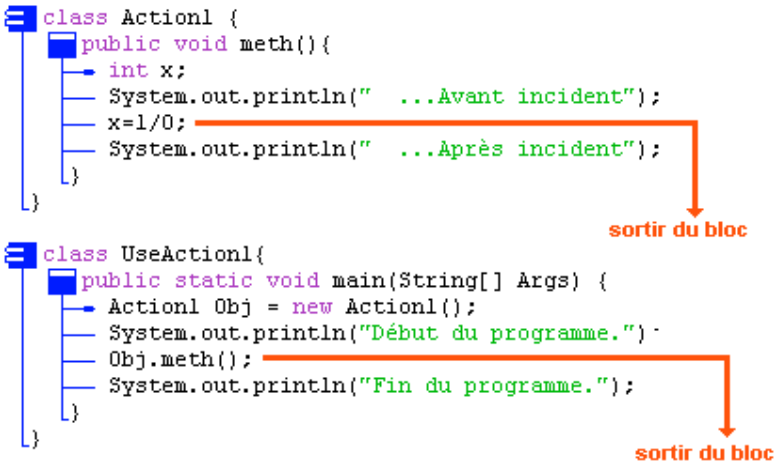
---- java UseAction1
Début du programme
** ...Avant incident**
java.lang.ArithmeticException : / by zero
---- : Exception in thread "main"

Que s'est-il passé ?

La méthode main :
- a instancié un objet Obj de classe Action1,;
- a affiché sur la console la phrase "Début du programme",
- a invoqué la méthode meth() de l'objet Obj,
- a affiché sur la console la phrase " ...Avant incident",
- a exécuté l'instruction "x = 1/0;"

Dès que l'instruction "x = 1/0;" a été exécutée celle-ci a provoqué un incident. **En fait une exception de la classe ArithmeticException a été "levée" (un objet de cette classe a été instancié) par la Java machine**.

La Java machine a arrêté le programme immédiatement à cet endroit parce qu'elle n'a pas trouvé de code d'interception de cette exception qui a été automatiquement levée :



Nous allons voir comment intercepter (on dit aussi "attraper" - to catch) cette exception afin de faire réagir notre programme afin qu'il ne s'arrête pas brutalement.

Programme avec gestion de l'exception

Java possède une instruction de gestion des exceptions, qui permet d'intercepter des exceptions dérivant de la classe Exception :

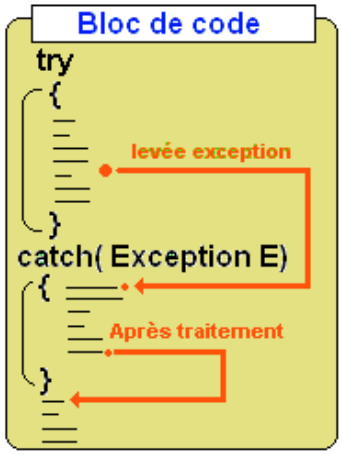
try ... catch

Syntaxe minimale d'un tel gestionnaire :

```
try {
        <lignes de code à protéger>
    }catch ( UneException E ) {
        <lignes de code réagissant à l'exception UneException >
    }
```

Le type **UneException** est obligatoirement une classe qui **hérite de la classe Exception**.

Schéma du fonctionnement d'un tel gestionnaire :



Le gestionnaire d'exception "déroute" l'exécution du programme vers le bloc d'interception catch qui traite l'exception (exécute le code contenu dans le bloc catch), puis renvoie et continue l'exécution du programme vers le code situé après le gestionnaire lui-même.

Principe de fonctionnement de l'interception

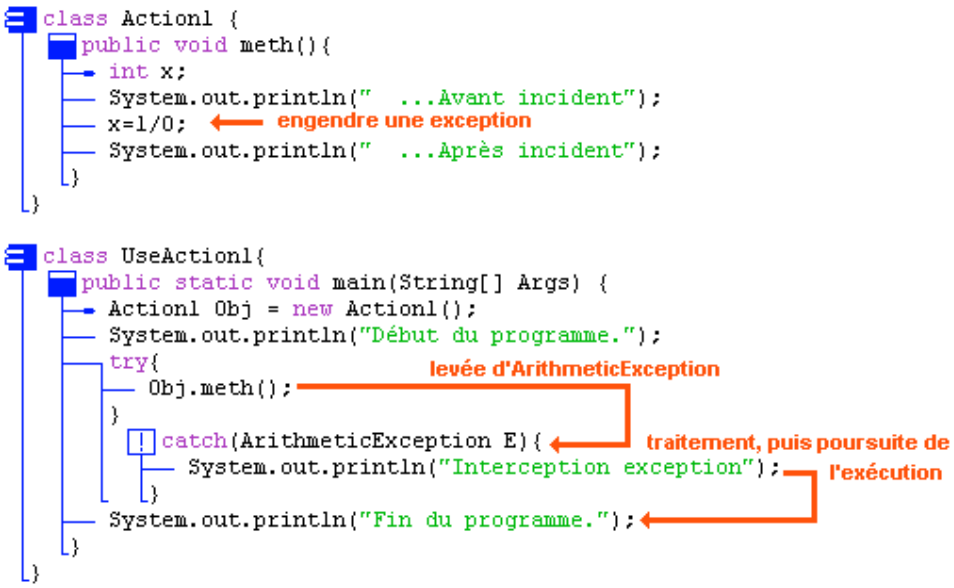
Dès qu'une **exception est levée** (instanciée), la Java machine **stoppe immédiatement** l'exécution normale du programme à la **recherche d'un gestionnaire** d'exception susceptible d'intercepter (saisir) et traiter cette exception. Cette recherche s'effectue à partir du **bloc englobant** et se poursuit sur les blocs plus englobants si aucun gestionnaire de **cette exception** n'a été trouvé.

Soit le même programme Java que précédemment, contenant un incident d'exécution (une division par zéro dans l'instruction x = 1/0;). Cette fois nous allons gérer l'incident grâce à un gestionnaire d'exception **try..catch** dans le bloc englobant immédiatement supérieur.

Programme sans traitement de l'incident :

```
class Action1 {
    public void meth(){
        int x;
        System.out.println("  ...Avant incident");
        x=1/0;
        System.out.println("  ...Après incident");
    }
}

class UseAction1{
    public static void main(String[] Args) {
        Action1 Obj = new Action1();
        System.out.println("Début du programme.");
        Obj.meth();
        System.out.println("Fin du programme.");
    }
}
```

Programme avec traitement de l'incident par try...catch :



Ci-dessous l'affichage obtenu sur la console lors de l'exécution de ce programme :

---- java UseAction1
Début du programme.
...Avant incident
Interception exception
Fin du programme.
---- : operation complete.

- Nous remarquons donc que la Java machine a donc bien exécuté le code d'interception situé dans le corps du "**catch** (ArithmeticException E){...}" et a poursuivi l'exécution normale après le gestionnaire.
- Le gestionnaire d'exception se situe dans la méthode main (code englobant) qui appelle la méthode meth() qui lève l'exception.

Interception d'exceptions hiérarchisées

Interceptions de plusieurs exceptions

Dans un gestionnaire try...catch, il est en fait possible d'intercepter plusieurs types d'exceptions différentes et de les traiter.

Ci-après nous montrons la syntaxe d'un tel gestionnaire qui fonctionne comme un selecteur ordonné, ce qui signifie qu'**une seule clause d'interception est exécutée**.

Dès qu'une exception intervient dans le < bloc de code à protéger>, la Java machine scrute séquentiellement toutes les clauses **catch** de la première jusqu'à la nième. Si l'exception

actuellement levée est d'un des types présents dans la liste des clauses le traitement associé est effectué, la scrutation est abandonnée et le programme poursuit son exécution après le gestionnaire.

```
try {
 < bloc de code à protéger>
}
catch ( TypeException1 E ) { <Traitement TypeException1 > }
catch ( TypeException2 E ) { <Traitement TypeException2 > }
  .....
 catch ( TypeExceptionk E ) { <Traitement TypeExceptionk > }
```

Où TypeException1, TypeException12, ... , TypeExceptionk sont des classes d'exceptions obligatoirement toutes **distinctes.**

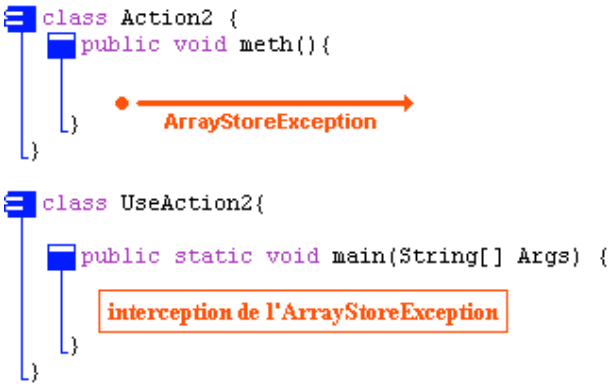
Seule une seule clause **catch** (TypeException E) {...}est exécutée (celle qui correspond au bon type de l'objet d'exception instancié).

Exemple théorique :

Supposons que la méthode meth() de la classe Action2 puisse lever trois types différents d'exceptions: ArithmeticException, ArrayStoreException, ClassCastException.

Notre gestionnaire d'exceptions est programmé pour intercepter l'une de ces 3 catégories. Nous figurons ci-dessous les trois schémas d'exécution correspondant chacun à la levée (l'instanciation d'un objet) d'une exception de l'un des trois types et son interception :

Interception d'une ArrayStoreException :



source java :

```
class Action2 {
    public void meth(){
        // une exception est levée ...
    }
}
```

ArrayStoreException

```
class UseAction2{
    public static void main(String[] Args) {
        Action1 Obj = new Action1();
        System.out.println("Début du programme.");
        try{
            Obj.meth();
        }
        catch(ArithmeticException E){
            System.out.println("Interception ArithmeticException");
        }
        catch(ArrayStoreException E){
            System.out.println("Interception ArrayStoreException");
        }
        catch(ClassCastException E){
            System.out.println("Interception ClassCastException");
        }
        System.out.println("Fin du programme.");
    }
}
```

ArrayStoreException

poursuite du programme

Il en est de même pour les deux autres types d'exception.

Interception d'une ClassCastException :

```
class Action2 {
    public void meth(){
        // une exception est levée ...
    }
}
```

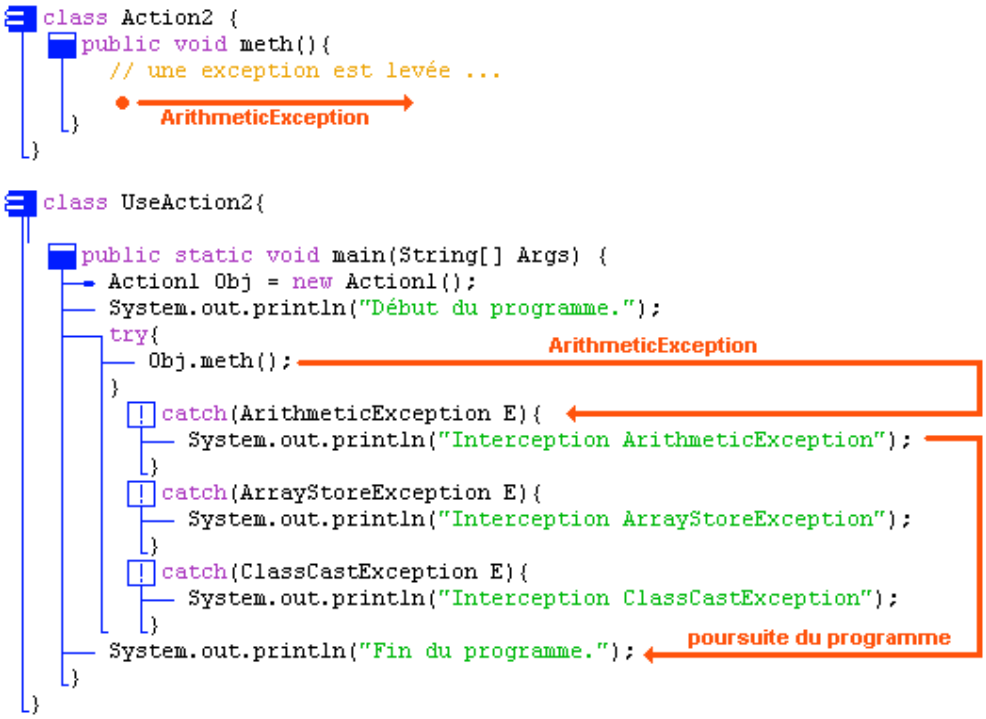
ClassCastException

```
class UseAction2{
    public static void main(String[] Args) {
        Action1 Obj = new Action1();
        System.out.println("Début du programme.");
        try{
            Obj.meth();
        }
        catch(ArithmeticException E){
            System.out.println("Interception ArithmeticException");
        }
        catch(ArrayStoreException E){
            System.out.println("Interception ArrayStoreException");
        }
        catch(ClassCastException E){
            System.out.println("Interception ClassCastException");
        }
        System.out.println("Fin du programme.");
    }
}
```

ClassCastException

poursuite du programme

Interception d'une ArithmeticException :



Ordre d'interception d'exceptions hiérarchisées

Dans un gestionnaire **try...catch** comprenant plusieurs clauses, la recherche de la clause catch contenant le traitement de la classe d'exception appropriée, s'effectue séquentiellement dans l'ordre d'écriture des lignes de code.

Soit le pseudo-code java suivant :

```
try {
  < bloc de code à protéger générant un objet exception>
}
catch ( TypeException1 E ) { <Traitement TypeException1 > }
catch ( TypeException2 E ) { <Traitement TypeException2 > }
  .....
catch ( TypeExceptionk E ) { <Traitement TypeExceptionk > }
```

La recherche va s'effectuer comme si le programme contenait des **if...else if...** imbriqués :

```
if (<Objet exception> instanceof TypeException1)  { <Traitement TypeException1 > }
else if (<Objet exception> instanceof TypeException2)  { <Traitement TypeException2 > }
...
else if (<Objet exception> instanceof TypeExceptionk)  { <Traitement TypeExceptionk > }
```

Les tests sont effectués sur l'appartenance de l'objet d'exception à une classe à l'aide de l'opérateur **instanceof**.

Signalons que l'opérateur **instanceof** agit sur une classe et ses classes filles (sur une hérarchie de classes), c'est à dire que tout objet de classe TypeExceptionX est aussi considéré comme un objet de classe parent au sens du test d'appartenance en particulier cet objet de classe TypeExceptionX est aussi considéré objet de classe Exception qui est la classe mère de toutes les exceptions Java.

Le test d'appartenance de classe dans la recherche d'une clause **catch** fonctionne d'une façon identique à l'opérateur **instanceof** dans les if...else

On choisira donc, lorsqu'il y a une hiérarchie entre les exceptions à intercepter, de placer le code de leurs gestionnaires dans l'ordre inverse de la hiérarchie.

Exemple : Soit une hiérarchie d'exceptions dans java

```
java.lang.Exception
  |
  +--java.lang.RuntimeException
        |
        +--java.lang.ArithmeticException
        |
        +--java.lang.ArrayStoreException
        |
        +--java.lang.ClassCastException
```

Soit le modèle de gestionnaire d'interception déjà fourni plus haut :

```
try {
  < bloc de code à protéger générant un objet exception>
}
catch ( ArithmeticException E ) { <Traitement ArithmeticException> }
catch ( ArrayStoreException E ) { <Traitement ArrayStoreException> }
catch ( ClassCastException E ) { <Traitement ClassCastException> }
```

Supposons que nous souhaitions intercepter une quatrième classe d'exception, par exemple une **RuntimeException,** nous devons rajouter une clause :

```
 catch ( RuntimeException E ) { <Traitement RuntimeException>}
```

Insérons cette clause en premier dans la liste des clauses d'interception :

```
class UseAction2{

   public static void main(String[] Args) {
      Action1 Obj = new Action1();
      System.out.println("Début du programme.");
      try{
         Obj.meth();
      }
        catch(RuntimeException E){
           System.out.println("Interception RuntimeException");
        }
        catch(ArithmeticException E){
           System.out.println("Interception ArithmeticException");
        }
        catch(ArrayStoreException E){
           System.out.println("Interception ArrayStoreException");
        }
        catch(ClassCastException E){
           System.out.println("Interception ClassCastException");
        }
      System.out.println("Fin du programme.");
   }
}
```

Nous lançons ensuite la compilation de cette classe et nous obtenons un message d'erreur :

```
class UseAction2{

   public static void main(String[] Args) {
      Action1 Obj = new Action1();
      System.out.println("Début du programme.");
      try{
         Obj.meth();
      }
        catch(RuntimeException E){
           System.out.println("Interception RuntimeException");
        }
        catch(ArithmeticException E){
           System.out.println("Interception ArithmeticException");
        }
        catch(ArrayStoreException E){
           System.out.println("Interception ArrayStoreException");
        }
        catch(ClassCastException E){
           System.out.println("Interception ClassCastException");
        }
      System.out.println("Fin du programme.");
   }
}
```

Résultats de l'exécution :

---- java UseAction2

UseAction2.java:19: exception java.lang.ArithmeticException has already been caught

catch(ArithmeticException E){

^

UseAction2.java:22: exception java.lang.ArrayStoreException has already been caught

```
        catch(ArrayStoreException E){
        ^
UseAction2.java:25: exception java.lang.ClassCastException has already been caught
        catch(ClassCastException E){
        ^
3 errors
```

Le compilateur proteste à partir de la clause **catch** (ArithmeticException E)en nous indiquant que l'exception est déjà interceptée et ceci trois fois de suite.

Que s'est-il passé ?

Le fait de placer en premier la clause **catch** (RuntimeException E) chargée d'intercepter les exceptions de classe RuntimeException implique que n'importe quelle exception héritant de RuntimeException comme par exemple ArithmeticException, est considérée comme une RuntimeException. Dans un tel cas cette ArithmeticException est interceptée par la clause **catch** (RuntimeException E) mais elle **n'est jamais** interceptée par la clause **catch** (ArithmeticException E).

Le seul endroit où le compilateur Java acceptera l'écriture de la clause **catch** (RuntimeException E) se situe **à la fin de la liste des clauses** catch. Ci-dessous l'écriture d'un programme correct :

```
class UseAction2{
  public static void main(String[] Args) {
    Action1 Obj = new Action1();
    System.out.println("Début du programme.");
    try{
      Obj.meth();
    }
      catch(ArithmeticException E){
        System.out.println("Interception ArithmeticException");
      }
      catch(ArrayStoreException E){
        System.out.println("Interception ArrayStoreException");
      }
      catch(ClassCastException E){
        System.out.println("Interception ClassCastException");
      }
      catch(RuntimeException E){
        System.out.println("Interception RuntimeException");
      }
    System.out.println("Fin du programme.");
  }
}
```

La classe parent doit être placée après ses classes filles

Dans ce cas la recherche séquentielle dans les clauses permettra le filtrage correct des classes filles puis ensuite le filtrage des classes mères.

On choisira donc, lorsqu'il y a une hiérarchie entre les exceptions à intercepter, de placer le code de leurs clauses dans l'ordre inverse de la hiérarchie.

Redéclenchement d'une exception : throw

Il est possible de déclencher soi-même des exceptions en utilisant l'instruction **throw**, voir même de déclencher des exceptions personnalisées ou non.

Déclenchement manuel d'une exception existante

La Java machine peut déclencher une exception automatiquement comme dans l'exemple de la levée d'une ArithmeticException lors de l'exécution de l'instruction "x = 1/0 ;".

La Java machine peut aussi lever (déclencher) une exception à votre demande suite à la rencontre d'une instruction **throw**. Le programme qui suit lance une ArithmeticException avec le message "Mauvais calcul !" dans la méthode **meth()** et intercepte cette exception dans le bloc englobant **main**. Le traitement de cette exception consiste à afficher le contenu du champ message de l'exception grâce à la méthode **getMessage()** :

```
class Action3 {
    public void meth(){
        int x=0;
        System.out.println("  ...Avant incident");
        if (x==0)
            throw new ArithmeticException("Mauvais calcul !");
        System.out.println("  ...Après incident");
    }
}

class UseAction3{
    public static void main(String[] Args) {
        Action3 Obj = new Action3();
        System.out.println("Début du programme.");
        try{
            Obj.meth();
        }
          catch(ArithmeticException E){
            System.out.println("Interception exception : "+E.getMessage());
          }
        System.out.println("Fin du programme.");
    }
}
```

Résultats de l'exécution :
---- java UseAction3
Début du programme.
...Avant incident
Interception exception : Mauvais calcul !
Fin du programme.
---- : operation complete.

Déclenchement manuel d'une exception personnalisée

Pour une exception personnalisée, le mode d'action est strictement identique, il vous faut seulement auparavant créer une nouvelle classe **héritant obligatoirement de la classe Exception** ou de n'importe laquelle de ses sous-classes.

Reprenons le programme précédent et créons une classe d'exception que nous nommerons ArithmeticExceptionPerso héritant de la classe des ArithmeticException puis exécutons ce programme :

```
class ArithmeticExceptionPerso extends ArithmeticException{
    ArithmeticExceptionPerso(String s){
        super(s);
    }
}

class Action3 {
    public void meth(){
        int x=0;
        System.out.println("  ...Avant incident");
        if (x==0)
            throw new ArithmeticExceptionPerso("Mauvais calcul !");
        System.out.println("  ...Après incident");
    }
}

class UseAction3{
    public static void main(String[] Args) {
        Action3 Obj = new Action3();
        System.out.println("Début du programme.");
        try{
            Obj.meth();
        }
            catch(ArithmeticExceptionPerso E){
                System.out.println("Interception exception : "+E.getMessage());
            }
        System.out.println("Fin du programme.");
    }
}
```

Résultats de l'exécution :
---- java UseAction3
Début du programme.
...Avant incident
Interception exception : Mauvais calcul !
Fin du programme.
---- : operation complete.

L'exécution de ce programme est identique à celle du programme précédent, notre exception fonctionne bien comme celle de Java.

Exception vérifiée ou non

La majorité des exceptions de Java font partie du package java.lang. Certaines exceptions sont tellement courantes qu'elles ont été rangées par Sun (concepteur de Java) dans une catégorie dénommée la catégorie des exceptions **implicites** ou **non vérifiées** (unchecked)**,** les autres sont dénommées exceptions **explici**tes ou **vérifiées** (checked) selon les auteurs.

Une exception **non vérifiée** (implicite) est une classe dérivant de l'une des deux classes **Error** ou **RuntimeException** :

```
java.lang.Object
  |
  +--java.lang.Throwable
       |
       +--java.lang.Error
       |
       +--java.lang.Exception
            |
            +--java.lang.RuntimeException
```

Toutes les exceptions vérifiées ou non fonctionnent de la même manière, la différence se localise dans la syntaxe à adopter dans une méthode propageant l'un ou l'autre genre d'exception.

Nous pouvons déjà dire que pour les exceptions non vérifiées, il n'y a aucune contrainte syntaxique pour propager une exception d'une méthode vers un futur bloc englobant. La meilleure preuve de notre affirmation est qu'en fait nous n'avons utilisé jusqu'ici que des exceptions non vérifiées, plus précisément des exceptions dérivant de RuntimeException et que nous n'avons utilisé aucune syntaxe spéciale dans la méthode meth() pour indiquer qu'elle était susceptible de lever une exception :

```
+--java.lang.RuntimeException
       |
       +--java.lang.ArithmeticException
       |
       +--java.lang.ArrayStoreException
       |
       +--java.lang.ClassCastException
```

Il n'en est pas de même lorsque l'exception lancée (levée) dans la méthode meth() est une exception vérifiée. Le paragraphe suivant vous explique comment agir dans ce cas.

Méthode propageant une exception vérifiée : throw

Une méthode dans laquelle est levée une ou plusieurs exceptions vérifiées doit obligatoirement signaler au compilateur quelles sont les classes d'exceptions qu'elle laisse se propager sans traitement par un gestionnaire (propagation vers un bloc englobant).

Java dispose d'un spécificateur pour ce signalement : le mot clef **throws** suivi de la liste des noms des classes d'exceptions qui sont propagées.

Signature générale d'une méthode propageant des exceptions vérifiées

```
<modificateurs> <type> < identificateur>  ( <liste param formels> )
          throws < liste d'exceptions> {
     .....
}

Exemple :

 protected static void meth ( int x, char c ) throws IOException, ArithmeticException {
  .....
}
```

Nous allons choisir une classe parmi les très nombreuses d'exceptions vérifiées, notre choix se porte sur une classe d'exception très utilisée, la classe IOException qui traite de toutes les exceptions d'entrée-sortie. Le J2SE 1.4.2 donne la liste des classes dérivant de la classe IOException :

ChangedCharSetException, CharacterCodingException, CharConversionException, ClosedChannelException, EOFException, FileLockInterruptionException, FileNotFoundException, IIOException, InterruptedIOException, MalformedURLException, ObjectStreamException, ProtocolException, RemoteException, SocketException, SSLException, SyncFailedException, UnknownHostException, UnknownServiceException, UnsupportedEncodingException, UTFDataFormatException, ZipException.

Reprenons le programme écrit au paragraphe précédent concernant le déclenchement manuel d'une exception existante en l'appliquant à la classe d'exception existante IOException, cette classe étant incluse dans le package **java.io** et non dans le package **java.lang** importé implicitement, on doit ajouter une instruction d'importation du package java.io, puis exécutons le programme tel quel. Voici ce que nous obtenons :

```
import java.io.*;

class Action4 {
    public void meth(){
        int x=0;
        System.out.println("  ...Avant incident");
        if (x==0)
            throw new IOException("Problème d'E/S !");
        System.out.println("  ...Après incident");
    }
}
```

```
class UseAction4{
    public static void main(String[] Args) {
        Action4 Obj = new Action4();
        System.out.println("Début du programme.");
        try{
            Obj.meth();
        }
            catch(IOException E){
                System.out.println("Interception exception : "+E.getMessage());
            }
        System.out.println("Fin du programme.");
    }
}
```

Résultats de l'exécution :

---- java UseAction4

UseAction4.java:8: unreported exception java.io.IOException; must be caught or declared to be thrown

throw new IOException("Problème d'E/S !");

^

1 error

Que s'est-il passé ?

Le compilateur Java attendait de notre part l'une des deux seules attitudes suivantes :

Soit nous interceptons et traitons l'exception IOException à l'intérieur du corps de la méthode où elle a été lancée avec pour conséquence que le bloc englobant n'aura pas à traiter une telle exception puisque l'objet d'exception est automatiquement détruit dès qu'il a été traité. Le code ci-après implante cette première attitude :

```
import java.io.*;

class Action4 {
    public void meth(){
        int x=0;
        System.out.println("   ...Avant incident");
        try{
            if (x==0)
                throw new IOException("Problème d'E/S !");}
            catch(IOException E){
                System.out.println("Interception exception : "+E.getMessage());
            }

        System.out.println("  ...Après incident");
    }
}

class UseAction4{
    public static void main(String[] Args) {
        Action4 Obj = new Action4();
        System.out.println("Début du programme.");
        Obj.meth();
        System.out.println("Fin du programme.");
    }
}
```

Interception de l'exception dans la méthode

Appel ordinaire de la méthode dans le bloc englobant

Résultats de l'exécution :
---- java UseAction4
Début du programme.
...Avant incident
Interception exception : Problème d'E/S !
...Après incident
Fin du programme.

Soit nous interceptons et traitons l'exception IOException à l'intérieur du bloc englobant la méthode meth() qui a lancé l'exception, auquel cas il est obligatoire de signaler au compilateur que cette méthode meth() lance et propage une IOException non traitée. Le code ci-après implante la seconde attitude possible :

```
import java.io.*;

class Action4 {
  public void meth()throws IOException {
    int x=0;
    System.out.println("  ...Avant incident");
    if (x==0)
      throw new IOException("Problème d'E/S !");
    System.out.println("  ...Après incident");
  }
}

class UseAction4{
  public static void main(String[] Args) {
    Action4 Obj = new Action4();
    System.out.println("Début du programme.");
    try{
      Obj.meth();
    }
      catch(IOException E){
        System.out.println("Interception exception : "+E.getMessage());
      }
    System.out.println("Fin du programme.");
  }
}
```

signaler l'exception susceptible d'être propagée

Interception de l'exception dans le bloc englobant

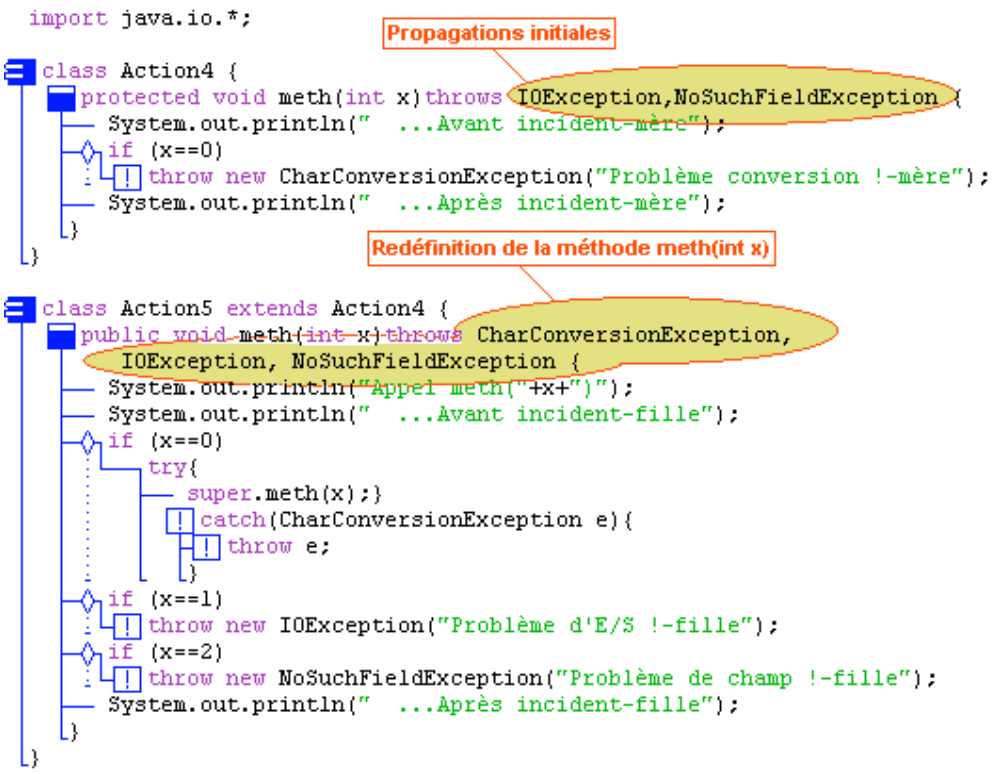
Résultats de l'exécution :
---- java UseAction4
Début du programme.
...Avant incident
Interception exception : Problème d'E/S !
Fin du programme.

Redéfinition d'une méthode propageant des exceptions vérifiées

Principe de base : la partie **throws** < **liste d'exceptions**> de la signature de la méthode qui redéfinit une méthode de la super-classe peut comporter moins de types d'exception. Elle ne peut pas propager plus de types ou des types différents de ceux de la méthode de la super-classe.

Ci-après un programme avec une super-classe **Action4** et une méthode **meth()**, qui est redéfinie

dans une classe fille nommée **Action5** :




```
import java.io.*;

class Action4 {
    protected void meth(int x)throws IOException,NoSuchFieldException {
        System.out.println("  ...Avant incident-mère");
        if (x==0)
            throw new CharConversionException("Problème conversion !-mère");
        System.out.println("  ...Après incident-mère");
    }
}

class Action5 extends Action4 {
    public void meth(int x)throws CharConversionException,
      IOException, NoSuchFieldException {
        System.out.println("Appel meth("+x+")");
        System.out.println("  ...Avant incident-fille");
        if (x==0)
            try{
                super.meth(x);}
            catch(CharConversionException e){
                throw e;
            }
        if (x==1)
            throw new IOException("Problème d'E/S !-fille");
        if (x==2)
            throw new NoSuchFieldException("Problème de champ !-fille");
        System.out.println("  ...Après incident-fille");
    }
}
```

Voici la hiérarchie des classes utilisées :

+--java.lang.Exception
|
+--java.io.IOException
| |
| +--java.io.CharConversionException
|
+--java.lang.NoSuchFieldException

Notez que la méthode meth de la super-classe propage IOException et NoSuchFieldException bien qu'elle ne lance qu'une exception de type IOException; ceci permettra la redéfinition dans la classe fille.

Voici pour terminer, un code source de classe utilisant la classe Action5 précédemment définie et engendrant aléatoirement l'un des 3 types d''exception :

```
class UseAction5{
  public static void main(String[] Args) {
    Action5 Obj = new Action5();
    System.out.println("Début du programme.");
    try{
      Obj.meth((int)(Math.random()*10)%3);
    }
      catch(CharConversionException E){
        System.out.println("Interception exception_0 : "+E.getMessage());
        }
      catch(IOException E){
        System.out.println("Interception exception_1 : "+E.getMessage());
        }
      catch(NoSuchFieldException E){
        System.out.println("Interception exception_2 : "+E.getMessage());
        }
    System.out.println("Fin du programme.");
  }
}
```

Analysez et comprenez bien le fonctionnement de ce petit programme.

Listing des 3 exécutions dans chacun des cas d'appel de la méthode meth :

Résultats de l'exécution
---- java UseAction5
Début du programme.
Appel meth(0)
...Avant incident-fille
...Avant incident-mère
Interception exception_0 : Problème conversion de caractère !-mère
Fin du programme.

Résultats de l'exécution
---- java UseAction5
Début du programme.
Appel meth(2)
...Avant incident-fille
Interception exception_2 : Problème de champ !-fille
Fin du programme.

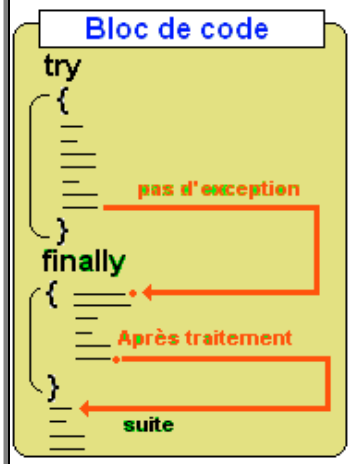
Résultats de l'exécution
---- java UseAction5
Début du programme.
Appel meth(1)
...Avant incident-fille
Interception exception_1 : Problème d'E/S !-fille
Fin du programme.

Clause finally

Supposons que nous soyons en présence d'un code contenant une éventuelle levée d'exception, mais supposons que quoiqu'il se passe nous désirions qu'un certain type d'action ait toujours lieu

(comme par exemple fermer un fichier qui a été ouvert auparavant). Il existe en Java une **clause spécifique optionnelle** dans la syntaxe des gestionnaires d'exception permettant ce type de réaction du programme, c'est la clause **finally.** Voici en pseudo Java une syntaxe de cette clause :

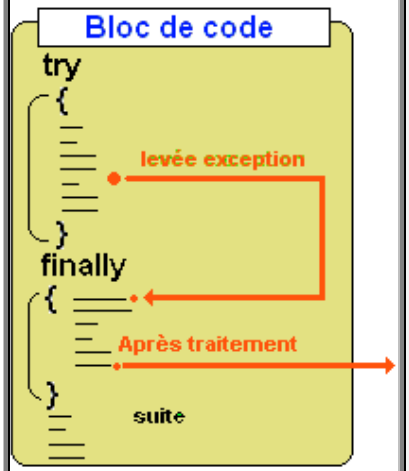
<Ouverture du fichier>
try {
< action sur fichier>
}
finally {
<fermeture du fichier>
}
.... suite



Fonctionnement dans le cas où rien n'a lieu

Si aucun incident ne se produit durant l'exécution du bloc < action sur fichier> :

l'exécution se poursuit à **l'intérieur du bloc finally par l'action** <fermeture du fichier> et **la suite** du code.



Fonctionnement si quelque chose se produit

Si un incident se produit durant l'exécution du bloc < action sur fichier> et qu'une exception est lancée:

malgré tout l'exécution se poursuivra à **l'intérieur du bloc finally par l'action** <fermeture du fichier>, puis **arrêtera** l'exécution du code dans le bloc.

La syntaxe Java autorise l'écriture d'une clause **finally** associée à plusieurs clauses **catch** :

```
try {
 <code à protéger>
}
catch (exception1 e ) {
  <traitement de l'exception1>}
catch (exception2 e ) {
  <traitement de l'exception2>}
...
finally {
  <action toujours effectuée>
}
```

Remarque :

> Si le code du bloc à protéger dans try...finally contient une instruction de rupture de séquence comme break, return ou continue, le code de la clause finally{...} est malgré tout exécuté avant la rupture de séquence.

Nous avons vu lors des définitions des itérations while, for et de l'instruction continue, que l'équivalence suivante entre un **for** et un **while** valide dans le cas général, était mise en défaut si le corps d'instruction contenait un **continue** (instruction forçant l'arrêt d'un tours de boucle et relançant l'itération suivante) :

*Equivalence incorrecte si Instr contient un **continue** :*

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

E*quivalence correcte même si Instr contient un **continue** :*

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

Le multi-threading

Java2

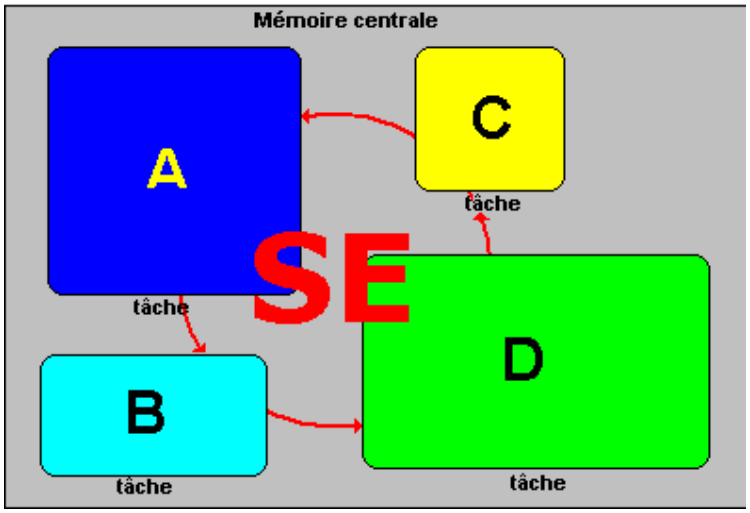
Le Multithreading

Nous savons que les ordinateurs fondés sur les principes d'une machine de Von Neumann, sont des machines séquentielles donc n'exécutant qu'une seule tâche à la fois. Toutefois, le gaspillage de temps engendré par cette manière d'utiliser un ordinateur (le processeur central passe l'écrasante majorité de son temps à attendre) a très vite été endigué par l'invention de systèmes d'exploitations de multi-programmation ou multi-tâches, permettant l'exécution "simultanée" de plusieurs tâches.

Dans un tel système, les différentes tâches sont exécutées sur une machine disposant d'**un seul processeur**, en apparence **en même temps** ou encore en **parallèle,** en réalité elles sont exécutées séquentiellement chacune à leur tour, ceci ayant lieu tellement vite pour notre conscience que nous avons l'impression que les programmes s'exécutent simultanément. Rappelons ici qu'une tâche est une application comme un traitement de texte, un navigateur internet, un jeu,... ou d'autres programmes spécifiques au système d'exploitation que celui-ci exécute.

Multitâche et thread

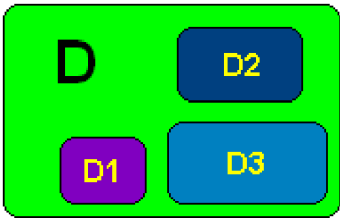
Le noyau du système d'exploitation SE, conserve en permanence le contrôle du temps d'exécution en distribuant cycliquement des tranches de temps (time-slicing) à chacune des applications A, B, C et D figurées ci-dessous. Dans cette éventualité, une application représente dans le système un **processus** :



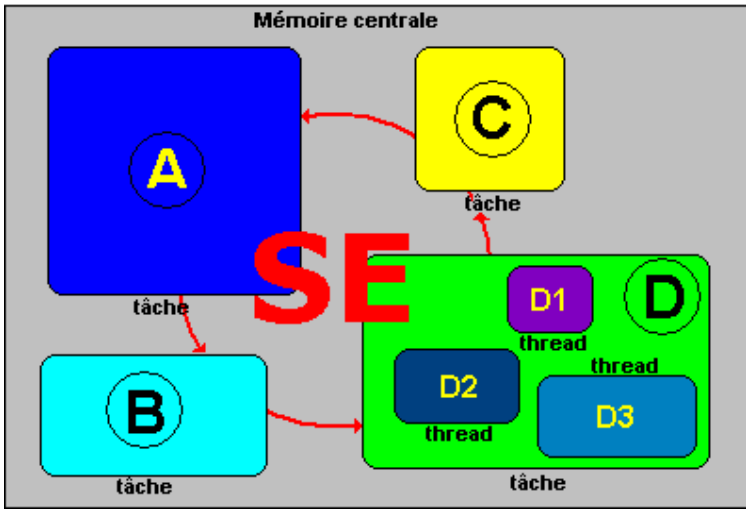
Rappelons la définition des **processus** donnée par A.Tanenbaum: un programme qui s'exécute et qui possède **son propre espace mémoire** : ses registres, ses piles, ses variables et son propre processeur virtuel (simulé en multi-programmation par la commutation entre processus effectuée par le processeur unique).

Thread
En fait, chaque processus peut lui-même fonctionner comme le système d'exploitation en lançant des sous-tâches internes au processus et par là même reproduire le fonctionnement de la multi-programmation. Ces sous-tâches sont nommées "flux d'exécution" ou **Threads.**

Ci-dessous nous supposons que l'application D exécute en même temps les 3 Threads D1, D2 et D3 :

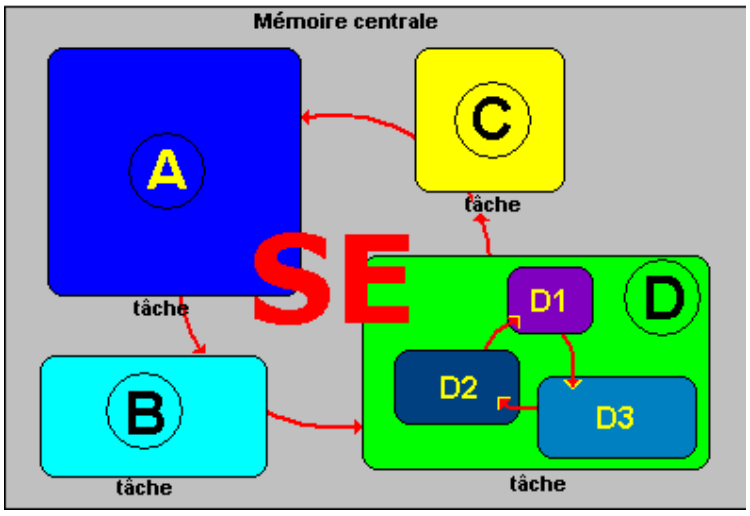


Reprenons l'exemple d'exécution précédent, dans lequel 4 processus s'exécutent "en même temps" et incluons notre processus D possédant 3 flux d'exécutions (threads) :



La commutation entre les threads d'un processus fonctionne de la même façon que la commutation entre les processus, chaque thread se voit alloué cycliquement, lorsque le processus D est exécuté une petite tranche de temps.

Le partage et la répartition du temps sont effectués **uniquement** par le système d'exploitation :



Multithreading et processus

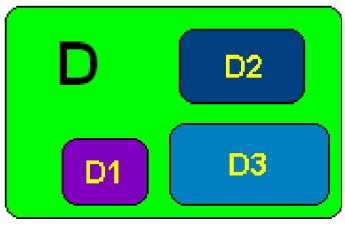
Définition :

La majorité des systèmes d'exploitation (Windows, Solaris, MacOs,...) supportent l'utilisation d'application contenant des threads, l'on désigne cette fonctionnalité sous le nom de **Multithreading.**

Différences entre threads et processus :

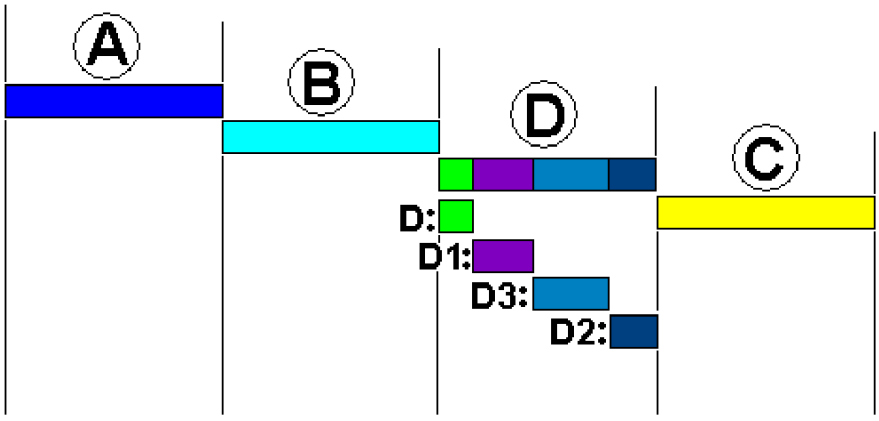
- Communication entre threads **plus rapide** que la communication entre processus,
- Les threads partagent un **même espace de mémoire** (de travail) entre eux,
- Les processus ont chacun un **espace mémoire personnel**.

Dans l'exemple précédent, figurons les processus A, B, C et le processus D avec ses threads dans un graphique représentant une tranche de temps d'exécution allouée par le système et supposée être la même pour chaque processus.



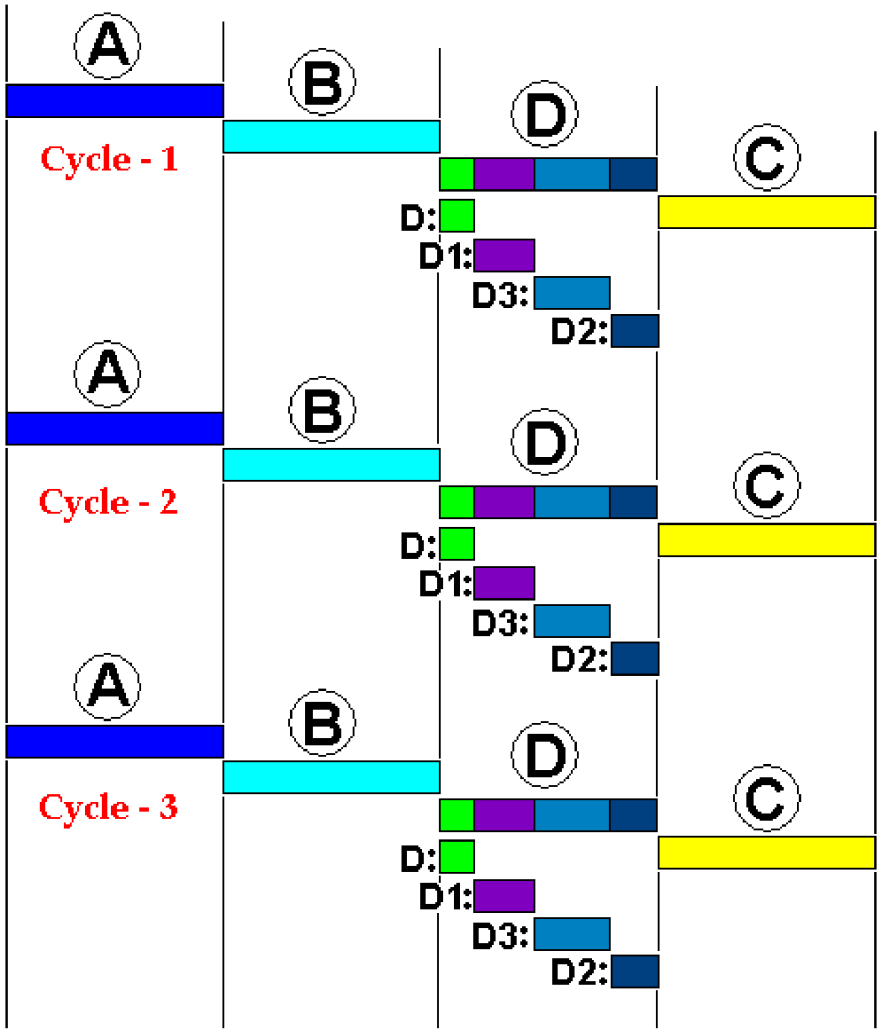
Le système ayant alloué le même temps d''exécution à chaque processus, lorsque par exemple le tour vient au processus D de s'exécuter dans sa tranche de temps, il exécutera une

petite sous-tranche pour D1, pour D2, pour D3 et attendra le prochain cycle. Ci-dessous un cycle d'exécution :



Tranches de temps allouées pendant l'exécution

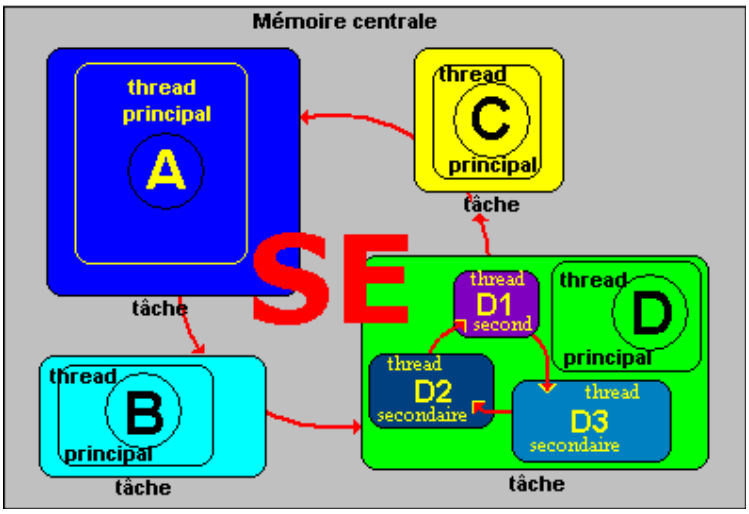
Voici sous les mêmes hypothèses de temps égal d'exécution alloué à chaque processus, le comportement de l'exécution sur 3 cycles consécutifs :



Les langages comme Delphi, Java et C# disposent chacun de classes permettant d'écrire et d'utiliser des threads dans vos applications.

Java autorise l'utilisation des threads

Lorsqu'un programme Java s'exécute en dehors d'une programmation de multi-threading, le processus associé comporte automatiquement un thread appelé **thread principal**. Un autre thread utilisé dans une application s'appelle un thread secondaire. Supposons que les quatre applications (ou tâches) précédentes A, B, C et D soient toutes des applications Java, et que D soit celle qui comporte trois threads secondaires D1, D2 et D3 "parallèlement" exécutés :



En Java c'est l'interface **Runnable** qui permet l'utilisation des threads dans la Java machine. Voici une déclaration de cette interface :

```
public interface Runnable {
   public void run( ) ;
}
```

Cette interface Runnable doit obligatoirement être implémentée par toute classe dont les instances seront exécutées par un thread. Une telle classe implémentant l'interface Runnable doit alors implémenter la méthode abstraite sans paramètre **public void** run( ) . Cette interface est conçue pour mettre en place un protocole commun à tous les objets qui souhaitent exécuter du code pendant que eux-mêmes sont actifs.

L'interface Runnable est essentiellement implantée par la classe **Thread** du package java.lang :

```
java.lang.Object
  |
  +--java.lang.Thread
```

Nous voyons maintenant comment concevoir une classe personnalisée dénommée **MaClasse** qui permettre l'exécution de ses instances dans des threads machines. Java vous offre deux façons différentes de décrire la classe **MaClasse** utilisant des threads :

- Soit par implémentation de l'interface **Runnable**.
- Soit par héritage de la classe **java.lang.Thread**

Interface Runnable
On effectue ce choix habituellement lorsque l'on veut utiliser une classe **ClassB** héritant d'une classe **ClassA** et que cette **ClassB** fonctionne comme un thread : il faudrait donc pouvoir hériter à la fois de la classe **ClassA** et de la classe **Thread,** ce qui est impossible, java ne connaît pas l'héritage multiple. En implémentant l'interface **Runnable** dans la classe **ClassB**, on rajoute à notre classe **ClassB** une nouvelle méthode **run ( )** (qui est en fait la seule méthode de l'interface Runnable).

Classe Thread
Dans le second cas l'héritage à partir de la classe Thread qui implémente l'interface **Runnable** et qui permet l'utilisation de méthodes de manipulation d'un thread (mise en attente, reprise,...).

Dans les deux cas, il vous faudra instancier un objet de classe Thread et vous devrez mettre ou invoquer le code à "exécuter en parallèle" dans le corps de la méthode run ( ) de votre classe .

L'interface Runnable

Première version Maclasse implémente l'interface Runnable :
vous héritez d'une seule méthode run ( ) que vous devez implémenter

Une classe dénommée MaClasse :

```
public interface Runnable {
    public void run( ) ;
}

class Thread implements Runnable{
    public void run( ){
    } ;
     //....
}

class MaClasse implements Runnable{
    public void run( ){
    } ;
     //....
}
```

Exemple de modèle :
les 2 threads principal et secondaire affichent chacun "en même temps" une liste de nombres de 1 à 100

```
public class MaClasse implements Runnable {
  Thread  UnThread ;
```

```
    MaClasse ( ) {
    //....initialisations éventuelles du constructeur de MaClasse
    UnThread = new Thread ( this , "thread secondaire" );
    UnThread.start( ) ;
    }

    public void run ( ) {
    //....actions du thread secondaire ici
    for ( int i1=1; i1<100; i1++)
            System.out.println(">>> i1 = "+i1);
    }
}
```

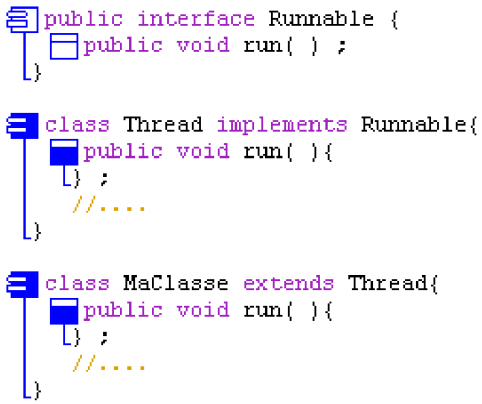
Pour utiliser cette classe contenant un thread secondaire, il est necessaire d'instancier un objet de cette classe :

```
class UtiliseMaclasse {
  public static void main(String[] x) {
     MaClasse tache1 = new MaClasse (  );
     /*.....le thread secondaire a été créé et activé
     le reste du code concerne le thread principal */
     for ( int i2 =1; i2<100;i2++)
            System.out.println(" i2 = "+i2);
  }
}
```

La classe java.lang.Thread

Deuxième version Maclasse dérive de la classe Thread :

vous héritez de toutes les méthodes de la classe Thread dont la méthode run( ) que vous devez redéfinir



Exemple de modèle :
les 2 threads principal et secondaire affichent chacun "en même temps" une liste de nombres

de 1 à 100

```
public class MaClasse extends Thread {

 MaClasse ( ) {
  //....initialisations éventuelles du constructeur de MaClasse
  this .start( ) ;
 }

 public void run ( ) {
 //....actions du thread secondaire ici
 for ( int i1=1; i1<100; i1++)
        System.out.println(">>> i1 = "+i1);
 }
}
```

Lorsque vous voulez utiliser cette classe contenant un thread secondaire, il vous faudra instancier un objet de cette classe de la même manière que pour la première version :

```
class UtiliseMaclasse {
  public static void main(String[] x) {
    MaClasse tache1 = new MaClasse (  );
    /*.....le thread secondaire a été créé et activé
   le reste du code concerne le thread principal */
   for ( int i2 =1; i2<100;i2++)
         System.out.println(" i2 = "+i2);
  }
}
```

Les deux versions par interface **Runnable** ou classe **Thread**, produisent le même résultat d'exécution. Chaque thread affiche séquentiellement sur la console ses données lorsque le système lui donne la main, ce qui donne un "mélange des deux affichages" :

Résultats d'exécution :

```
i2 = 1
i2 = 2
i2 = 3
i2 = 4
i2 = 5
i2 = 6
i2 = 7
i2 = 8
i2 = 9
i2 = 10
i2 = 11
i2 = 12
i2 = 13
i2 = 14
i2 = 15
i2 = 16
i2 = 17
i2 = 18
i2 = 19
i2 = 20
i2 = 21
i2 = 22
i2 = 23
i2 = 24
i2 = 25
```

```
i2 = 50
i2 = 51
i2 = 52
i2 = 53
i2 = 54
i2 = 55
i2 = 56
i2 = 57
i2 = 58
>>> i1 = 1
i2 = 59
>>> i1 = 2
i2 = 60
>>> i1 = 3
i2 = 61
>>> i1 = 4
i2 = 62
i2 = 63
i2 = 64
i2 = 65
i2 = 66
i2 = 67
i2 = 68
i2 = 69
i2 = 70
```

```
i2 = 95
i2 = 96
i2 = 97
i2 = 98
i2 = 99
>>> i1 = 5
>>> i1 = 6
>>> i1 = 7
>>> i1 = 8
>>> i1 = 9
>>> i1 = 10
>>> i1 = 11
>>> i1 = 12
>>> i1 = 13
>>> i1 = 14
>>> i1 = 15
>>> i1 = 16
>>> i1 = 17
>>> i1 = 18
>>> i1 = 19
>>> i1 = 20
>>> i1 = 21
>>> i1 = 22
>>> i1 = 23
>>> i1 = 24
```

```
>>> i1 = 50
>>> i1 = 51
>>> i1 = 52
>>> i1 = 53
>>> i1 = 54
>>> i1 = 55
>>> i1 = 56
>>> i1 = 57
>>> i1 = 58
>>> i1 = 59
>>> i1 = 60
>>> i1 = 61
>>> i1 = 62
>>> i1 = 63
>>> i1 = 64
>>> i1 = 65
>>> i1 = 66
>>> i1 = 67
>>> i1 = 68
>>> i1 = 69
>>> i1 = 70
>>> i1 = 71
>>> i1 = 72
>>> i1 = 73
>>> i1 = 74
```

i2 = 26 i2 = 27 i2 = 28 i2 = 29 i2 = 30 i2 = 31 i2 = 32 i2 = 33 i2 = 34 i2 = 35 i2 = 36 i2 = 37 i2 = 38 i2 = 39 i2 = 40 i2 = 41 i2 = 42 i2 = 43 i2 = 44 i2 = 45 i2 = 46 i2 = 47 i2 = 48 i2 = 49	i2 = 71 i2 = 72 i2 = 73 i2 = 74 i2 = 75 i2 = 76 i2 = 77 i2 = 78 i2 = 79 i2 = 80 i2 = 81 i2 = 82 i2 = 83 i2 = 84 i2 = 85 i2 = 86 i2 = 87 i2 = 88 i2 = 89 i2 = 90 i2 = 91 i2 = 92 i2 = 93 i2 = 94	>>> i1 = 25 >>> i1 = 26 >>> i1 = 27 >>> i1 = 28 >>> i1 = 29 >>> i1 = 30 >>> i1 = 31 >>> i1 = 32 >>> i1 = 33 >>> i1 = 34 >>> i1 = 35 >>> i1 = 36 >>> i1 = 37 >>> i1 = 38 >>> i1 = 39 >>> i1 = 40 >>> i1 = 41 >>> i1 = 42 >>> i1 = 43 >>> i1 = 44 >>> i1 = 45 >>> i1 = 46 >>> i1 = 47 >>> i1 = 48 >>> i1 = 49	>>> i1 = 75 >>> i1 = 76 >>> i1 = 77 >>> i1 = 78 >>> i1 = 79 >>> i1 = 80 >>> i1 = 81 >>> i1 = 82 >>> i1 = 83 >>> i1 = 84 >>> i1 = 85 >>> i1 = 86 >>> i1 = 87 >>> i1 = 88 >>> i1 = 89 >>> i1 = 90 >>> i1 = 91 >>> i1 = 92 >>> i1 = 93 >>> i1 = 94 >>> i1 = 95 >>> i1 = 96 >>> i1 = 97 >>> i1 = 98 >>> i1 = 99

---- operation complete.

Méthodes de base sur les objets de la classe Thread

Les méthodes static de la classe Thread travaillent sur le thread courant (*en l'occurrence le thread qui invoque la méthode static*) :

static int activeCount( )	renvoie le nombre total de threads actifs actullement.
static Thread currentThread( )	renvoie la référence de l'objet thread actuellement exécuté (thread courant)
static void dumpStack( )	Pour le debugging : une trace de la pile du thread courant
static int enumerate(Thread [] tarray)	Renvoie dans le tableau tarray[] les reférences de tous les threads actifs actullement.
static boolean interrupted( )	Teste l'indicateur d'interruption du thread courant, mais change cet interrupteur (utiliser plutôt la méthode d'instance **boolean** isInterrupted( )).
static void sleep(**long** millis, **int** nanos)	Provoque l'arrêt de l'exécution du thread courant pendant milis milisecondes et nanos nanosecondes.

static void sleep(**long** millis)	Provoque l'arrêt de l'exécution du thread courant pendant milis milisecondes.

Reprenons l'exemple ci-haut de l'affichage entre-mêlé de deux listes d'entiers de 1 à 100 :

```
class MaClasse2 extends Thread {

   MaClasse2 ( ) {
     this .start( ) ;
   }

   public void run ( ) {
   for ( int i1=1; i1<100; i1++)
          System.out.println(">>> i1 = "+i1);
   }
}

class UtiliseMaclasse2 {
  public static void main(String[] x) {
     // corps du thread principal
     MaClasse2 tache1 = new MaClasse (  ); // le thread secondaire
     for ( int i2 =1; i2<100;i2++)
          System.out.println(" i2 = "+i2);
  }
}
```

Arrêtons pendant une milliseconde l'exécution du thread principal : **Thread.sleep(1);** comme la méthode de classe sleep( ) propage une InterruptedException nous devons l'intercepter :

```
class UtiliseMaclasse2 {
   public static void main(String[] x) {
      MaClasse2 tache1 = new MaClasse2 (  );
   //.......le thread secondaire a été créé et activé le reste du
      try {
         Thread.sleep(1);
        }
          catch(InterruptedException e){
            System.out.println(e);
          }
      for(int i2=1;i2<100;i2++)
         System.out.println("i2 = "+i2);
   }
}
```

Ralenti de 1ms le thread principal

En **arrêtant pendant 1ms** le thread principal, nous laissons donc plus de temps au thread secondaire pour s'exécuter, les affichages ci-dessous le montrent :

Résultats d'exécution :

>>> i1 = 1 >>> i1 = 2 >>> i1 = 3	>>> i1 = 50 >>> i1 = 51 >>> i1 = 52	i2 = 13 i2 = 14 i2 = 15	>>> i1 = 95 >>> i1 = 96 >>> i1 = 97

```
>>> i1 = 4
>>> i1 = 5
>>> i1 = 6
>>> i1 = 7
>>> i1 = 8
>>> i1 = 9
>>> i1 = 10
>>> i1 = 11
>>> i1 = 12
>>> i1 = 13
>>> i1 = 14
>>> i1 = 15
>>> i1 = 16
>>> i1 = 17
>>> i1 = 18
>>> i1 = 19
>>> i1 = 20
>>> i1 = 21
>>> i1 = 22
>>> i1 = 23
>>> i1 = 24
>>> i1 = 25
>>> i1 = 26
>>> i1 = 27
>>> i1 = 28
>>> i1 = 29
>>> i1 = 30
>>> i1 = 31
>>> i1 = 32
>>> i1 = 33
>>> i1 = 34
>>> i1 = 35
>>> i1 = 36
>>> i1 = 37
>>> i1 = 38
>>> i1 = 39
>>> i1 = 40
>>> i1 = 41
>>> i1 = 42
>>> i1 = 43
>>> i1 = 44
>>> i1 = 45
>>> i1 = 46
>>> i1 = 47
>>> i1 = 48
>>> i1 = 49
>>> i1 = 53
>>> i1 = 54
>>> i1 = 55
>>> i1 = 56
>>> i1 = 57
>>> i1 = 58
>>> i1 = 59
>>> i1 = 60
>>> i1 = 61
>>> i1 = 62
>>> i1 = 63
>>> i1 = 64
>>> i1 = 65
>>> i1 = 66
>>> i1 = 67
>>> i1 = 68
>>> i1 = 69
>>> i1 = 70
>>> i1 = 71
>>> i1 = 72
>>> i1 = 73
>>> i1 = 74
>>> i1 = 75
>>> i1 = 76
>>> i1 = 77
>>> i1 = 78
>>> i1 = 79
>>> i1 = 80
>>> i1 = 81
i2 = 1
>>> i1 = 82
i2 = 2
>>> i1 = 83
i2 = 3
>>> i1 = 84
i2 = 4
>>> i1 = 85
i2 = 5
>>> i1 = 86
i2 = 6
i2 = 7
i2 = 8
i2 = 9
i2 = 10
i2 = 11
i2 = 12
i2 = 16
i2 = 17
i2 = 18
i2 = 19
i2 = 20
i2 = 21
i2 = 22
i2 = 23
i2 = 24
i2 = 25
i2 = 26
i2 = 27
i2 = 28
i2 = 29
i2 = 30
i2 = 31
i2 = 32
i2 = 33
i2 = 34
i2 = 35
i2 = 36
i2 = 37
i2 = 38
i2 = 39
i2 = 40
i2 = 41
i2 = 42
i2 = 43
i2 = 44
i2 = 45
i2 = 46
i2 = 47
i2 = 48
i2 = 49
i2 = 50
i2 = 51
>>> i1 = 87
i2 = 52
>>> i1 = 88
i2 = 53
>>> i1 = 89
i2 = 54
>>> i1 = 90
>>> i1 = 91
>>> i1 = 92
>>> i1 = 93
>>> i1 = 94
>>> i1 = 98
>>> i1 = 99
i2 = 55
i2 = 56
i2 = 57
i2 = 58
i2 = 59
i2 = 60
i2 = 61
i2 = 62
i2 = 63
i2 = 64
i2 = 65
i2 = 66
i2 = 67
i2 = 68
i2 = 69
i2 = 70
i2 = 71
i2 = 72
i2 = 73
i2 = 74
i2 = 75
i2 = 76
i2 = 77
i2 = 78
i2 = 79
i2 = 80
i2 = 81
i2 = 82
i2 = 83
i2 = 84
i2 = 85
i2 = 86
i2 = 87
i2 = 88
i2 = 89
i2 = 90
i2 = 91
i2 = 92
i2 = 93
i2 = 94
i2 = 95
i2 = 96
i2 = 97
i2 = 98
i2 = 99
```

---- operation complete.

Les méthodes d'instances de la classe Thread

Java permet d'arrêter (**stop**), de faire attendre (**sleep, yield**), d'interrompre (**interrupt**), de changer la priorité(**setPriority**), de détruire(**destroy**) des threads à travers les méthodes de la classe Thread.

Java permet à des threads de "communiquer" entre eux sur leur état (utiliser alors les méthodes : **join, notify, wait**).

Exercices Java2

ALGORITHMES A TRADUIRE EN JAVA

Algorithme

Calcul de la valeur absolue d'un nombre réel

Objectif : Ecrire un programme Java servant à calculer la valeur absolue d'un nombre réel x à partir de la définition de la valeur absolue. La valeur absolue du nombre réel x est le nombre réel |x| :

|x| = x , si x □□

|x| = -x si x < 0

Spécifications de l'algorithme :

```
lire( x );
si x □0 alors  écrire( '|x| =', x)
sinon  écrire( '|x| =', -x)
fsi
```

Implantation en Java

Ecrivez avec les deux instructions différentes "**if...else..**" et "**...?.. : ...**", le programme Java complet correspondant à l'affichage ci-dessous :

```
Entrez un nombre x = -45
  |x| = 45
```

Proposition de squelette de classe Java à implanter :

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

La méthode **main** calcule et affiche la valeur absolue.

Algorithme

Algorithme de résolution de l'équation du second degré dans R.

Objectif : On souhaite écrire un programme Java de résolution dans R de l'équation du second degré : $Ax^2 + Bx + C = 0$

Il s'agit ici d'un algorithme très classique provenant du cours de mathématique des classes du secondaire. L'exercice consiste essentiellement en la traduction immédiate

Spécifications de l'algorithme :

Algorithme Equation
  Entrée: A, B, C □ **Réels**
  Sortie: X1 , X2 □ **Réels**
  Local: □ □ **Réels**

début
 lire(A, B, C);
 Si A=0 **alors début**{A=0}
  Si B = 0 **alors**
   Si C = 0 **alors**
    écrire(R est solution)
   Sinon{C □ 0}
    écrire(pas de solution)
   Fsi
  Sinon {B □ 0}
   X1 □ C/B;
   écrire (X1)
  Fsi
 fin
 Sinon {A □ 0}**début**
  □ □ B^2 - 4*A*C ;
  Si □ < 0 **alors**
   écrire(pas de solution)
  Sinon {□ □0}
   Si □ = 0 **alors**
    X1 □ -B/(2*A);
    écrire (X1)
   Sinon{□ □ 0}
    X1 □ (-B +□□)/(2*A);
    X2 □ (-B - □□)/(2*A);

écrire(X1 , X2)
Fsi
Fsi
fin
Fsi
FinEquation

Implantation en Java

Ecrivez le programme Java qui est la traduction immédiate de cet algorithme dans le corps de la méthode main.

Proposition de squelette de classe Java à implanter :

```
class ApplicationEqua2 {
    public static void main(String[ ] args) {
        ……..
    }
}
```

Conseil :

On utilisera la méthode **static** sqrt(double x) de la classe **Math** pour calculer la racine carré d'un nombre réel :

□□ se traduira alors par : Math.sqrt(delta)

Algorithme
Calcul des nombres de Armstrong

Objectif : On dénomme nombre de Armstrong un entier naturel qui est égal à la somme des cubes des chiffres qui le composent.

***Exemple* :**

$153 = 1^3 + 5^3 + 3^3$

153 = 1 + 125 + 27, est un nombre de Armstrong.

Spécifications de l'algorithme :

On sait qu'il n'existe que 4 nombres de Armstrong, et qu'ils ont tous 3 chiffres (ils sont compris entre 100 et 500).
Si l'on qu'un tel nombre est écrit ijk (i chiffre des centaines, j chiffres des dizaines et k chiffres des unités), il suffit simplement d'envisager tous les nombres possibles en faisant varier les chiffres entre 0 et 9 et de tester si le nombre est de Armstrong.

Implantation en Java

Ecrivez le programme Java complet qui fournisse les 4 nombres de Armstrong :

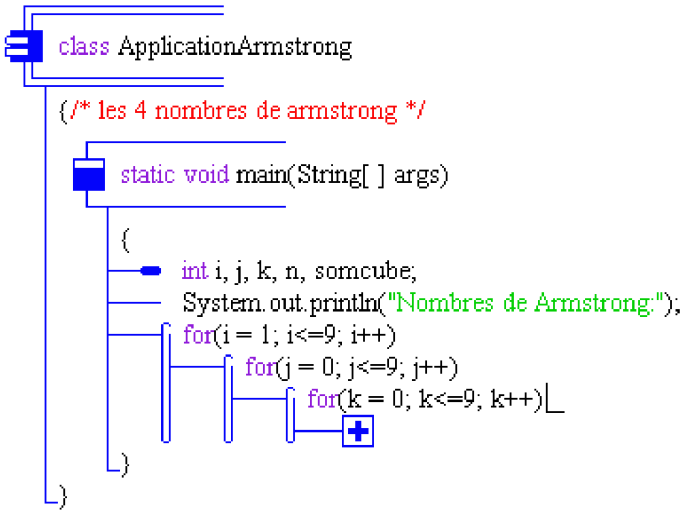
```
Nombres de Armstrong:
   153
   370
   371
   407
```

Proposition de squelette de classe Java à implanter :

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

La méthode **main** calcule et affiche les nombres de Armstrong.

Squelette plus détaillé de la classe Java à implanter :



Algorithme
Calcul de nombres parfaits

Objectif : On souhaite écrire un programme java de calcul des n premiers nombres parfaits. Un nombre est dit parfait s'il est égal à la somme de ses diviseurs, 1 compris.

***Exemple* : 6 = 1+2+3 , est un nombre parfait.**

Spécifications de l'algorithme :

l'algorithme retenu contiendra deux boucles imbriquées. Une boucle de comptage des nombres parfaits qui s'arrêtera lorsque le décompte sera atteint, la boucle interne ayant vocation à calculer tous les diviseurs du nombre examiné d'en faire la somme puis de tester l'égalité entre cette somme et le nombre.

```
Algorithme Parfait
  Entrée: n  N
  Sortie: nbr  N
  Local: somdiv, k, compt   N

    début
     lire(n);
     compt    0;
     nbr    2;
    Tantque(compt < n) Faire
      somdiv    1;
      Pour k    2 jusquà nbr-1 Faire
        Si reste(nbr par k) = 0 Alors // k  divise nbr
          somdiv    somdiv + k
        Fsi
      Fpour ;
      Si somdiv = nbr Alors
        ecrire(nbr) ;
        compt    compt+1;
      Fsi;
      nbr    nbr+1
    Ftant
   FinParfait
```

Implantation en Java

Ecrivez le programme Java complet qui produise le dialogue suivant à l'écran (les caractères gras représentent ce qui est écrit par le programme, les italiques ce qui est entré au clavier) :

```
Entrez combien de nombre parfaits : 4
    6 est un nombre parfait
    28 est un nombre parfait
    496 est un nombre parfait
    8128 est un nombre parfait
```

Proposition de squelette de classe Java à implanter :

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

La méthode **main** calcule et affiche les nombres parfaits

Algorithme

Calcul du pgcd de 2 entiers (méthode Euclide)

Objectif : On souhaite écrire un programme de calcul du pgcd de deux entiers non nuls, en Java à partir de l'algorithme de la méthode d'Euclide. Voici une spécification de l'algorithme de calcul du PGCD de deux nombres (entiers strictement positifs) a et b, selon cette méthode :

Spécifications de l'algorithme :

```
Algorithme Pgcd
 Entrée: a,b  N* x N*
 Sortie: pgcd  N
 Local: r,t  N x N

début
 lire(a,b);
 Si ba Alors
   t  a ;
   a  b ;
   b  t
 Fsi;
 Répéter
   r  a mod b ;
   a  b ;
   b  r
 jusquà r = 0;
 pgcd  a;
 ecrire(pgcd)
FinPgcd
```

Implantation en Java

Ecrivez le programme Java complet qui produise le dialogue suivant à l'écran (les caractères gras représentent ce qui est écrit par le programme, les italiques ce qui est entré au clavier) :

Entrez le premier nombre : *21*
Entrez le deuxième nombre : *45*
Le PGCD de 21 et 45 est : 3

Proposition de squelette de classe Java à implanter :

```
class ApplicationEuclide {
    public static void main(String[ ] args) {
       ……..
    }
    static int pgcd (int a, int b) {
       ……..
    }
 }
```

La méthode **pgcd** renvoie le pgcd des deux entiers p et q .

Algorithme

Calcul du pgcd de 2 entiers (méthode Egyptienne)

Objectif : On souhaite écrire un programme de calcul du pgcd de deux entiers non nuls, en Java à partir de l'algorithme de la méthode dite "égyptienne " Voici une spécification de l'algorithme de calcul du PGCD de deux nombres (entiers strictement positifs) p et q, selon cette méthode :

Spécifications de l'algorithme :

```
Lire (p, q ) ;

Tantque p □ q faire
    Si p > q alors

      p □  p – q

    sinon

      q □  q – p

    FinSi
FinTant;
Ecrire(  " PGCD = " , p )
```

Implantation en Java

Ecrivez le programme Java complet qui produise le dialogue suivant à l'écran (les caractères gras représentent ce qui est écrit par le programme, les italiques ce qui est entré au clavier) :

Entrez le premier nombre : *21*
Entrez le deuxième nombre : *45*
Le PGCD de 21 et 45 est : 3

Proposition de squelette de classe Java à implanter :

```
class ApplicationEgyptien {
    public static void main(String[ ] args) {
        ……..
    }
    static int pgcd (int p, int q) {
        ……..
    }
}
```

La méthode **pgcd** renvoie le pgcd des deux entiers p et q .

Algorithme

Calcul de nombres premiers (boucles **while** et **do**…**while**)

Objectif : On souhaite écrire un programme Java de calcul et d'affichage des n premiers nombres premiers. Un nombre entier est premier s'il n'est divisible que par 1 et par lui-même On opérera une implantation avec des boucles while et do...while.

Exemple : 37 est un nombre premier

Spécifications de l'algorithme :

```
Algorithme Premier
  Entrée: n  N
  Sortie: nbr   N
  Local: Est_premier  {Vrai , Faux}
         divis,compt  N²;

    début
     lire(n);
     compt    1;
     ecrire(2);
     nbr    3;
     Tantque(compt < n) Faire
      divis    3;
      Est_premier    Vrai;
      Répéter
        Si reste(nbr par divis) = 0 Alors
         Est_premier    Faux
        Sinon
         divis    divis+2
        Fsi
      jusquà (divis > nbr / 2)ou (Est_premier=Faux);
      Si Est_premier =Vrai Alors
        ecrire(nbr);
        compt    compt+1
      Fsi;
      nbr    nbr+1
     Ftant
    FinPremier
```

Implantation en Java

Ecrivez le programme Java complet qui produise le dialogue suivant à l'écran (les caractères gras représentent ce qui est écrit par le programme, les italiques ce qui est entré au clavier) :

Combien de nombres premiers : *5*
2
3
5
7
11

Proposition de squelette de classe Java à implanter avec une boucle while et une boucle do...while imbriquée :

On étudie la primalité de tous les nombres systématiquement

```
class ApplicationComptPremiers1 {
    public static void main(String[ ] args) {
      .....
  while (compt < n)
  {
      divis=2 ;
      Est_premier=true;
      do
        {
      if(Est_premier)
        {
      nbr++ ;
  }
      ......
    }
  }
```

La méthode **main** affiche la liste des nombres premiers demandés.

Algorithme

Calcul de nombres premiers (boucles **for**)

Objectif : On souhaite écrire un programme Java de calcul et d'affichage des n premiers nombres premiers. Un nombre entier est premier s'il n'est divisible que par 1 et par lui-même. On opérera une implantation avec des boucles for imbriquées.

Exemple : 19 est un nombre premier

Spécifications de l'algorithme : (o*n étudie la primalité des nombres uniquement impairs)*

```
Algorithme Premier
  Entrée: n  N
  Sortie: nbr   N
  Local: Est_premier  {Vrai , Faux}
         divis,compt  N²;

   début
    lire(n);
    compt    1;
    ecrire(2);
    nbr    3;
    Tantque(compt < n) Faire
     divis    3;
     Est_premier    Vrai;
     Répéter
       Si reste(nbr par divis) = 0 Alors
        Est_premier    Faux
       Sinon
        divis    divis+2
       Fsi
     jusquà (divis > nbr / 2)ou (Est_premier=Faux);
     Si Est_premier =Vrai Alors
       ecrire(nbr);
       compt    compt+1
     Fsi;
     nbr    nbr+2 // nbr impairs
    Ftant
   FinPremier
```

Implantation en Java

Ecrivez le programme Java complet qui produise le dialogue suivant à l'écran (les caractères gras représentent ce qui est écrit par le programme, les italiques ce qui est entré au clavier) :

Combien de nombres premiers : *5*
2
3
5
7
11

Proposition de squelette de classe Java à implanter avec deux boucles **for** imbriquées :

On étudie la primalité des nombres uniquement impairs

```
class ApplicationComptPremiers2 {
    public static void main(String[ ] args) {
      .....
for( nbr = 3; compt < max; nbr += 2 )
{ Est_premier = true;
   for (divis = 2; divis<= nbr/2; divis++ )
   if(Est_premier)
}
      ......
    }
  }
```

La méthode **main** affiche la liste des nombres premiers demandés.

Le fait de n'étudier la primalité que des nombres impairs accélère la vitesse d'exécution du programme, il est possible d'améliorer encore cette vitesse en ne cherchant que les diviseurs dont le carré est inférieur au nombre (test : **jusquà** ($divis^2 > nbr$)**ou** (Est_premier=Faux))

Algorithme
Calcul du nombre d'or

Objectif : On souhaite écrire un programme Java qui calcule le nombre d'or utilisé par les anciens comme nombre idéal pour la sculpture et l'architecture. Si l'on considère deux suites numériques (U) et (V) telles que pour n strictement supérieur à 2 :

$$U_n = U_{n-1} + U_{n-2}$$

et

$$V_n = U_n / U_{n-1}$$

On montre que la suite (V) tend vers une limite appelée nombre d'or (nbr d'Or = 1,6180339887498948482045868343656 4).

Spécifications de l'algorithme :

n, U_n, U_{n1}, U_{n2} : sont des entiers naturels
V_n, V_{n1}, □ : sont des nombres réels

lire(□); *// précision demandée*
  U_{n2} □ 1;
  U_{n1} □ 2;
  V_{n1} □ 2;
  n □ 2; *// rang du terme courant*
Itération
  n □ n + 1;
  U_n □ $U_{n1} + U_{n2}$;
  V_n □ U_n / U_{n1} ;
  si $|V_n - V_{n1}|$ □ □**alors** Arrêt de la boucle ; *// la précision est atteinte*
  sinon
    U_{n2} □ U_{n1} ;
    U_{n1} □ U_n ;
    V_{n1} □ V_n ;
  fsi
fin Itération
ecrire (V_n , n);

Ecrire un programme fondé sur la spécification précédente de l'algorithme du calcul du nombre d'or. Ce programme donnera une valeur approchée avec une précision fixée de □du

nombre d'or. Le programme indiquera en outre le rang du dernier terme de la suite correspondant.

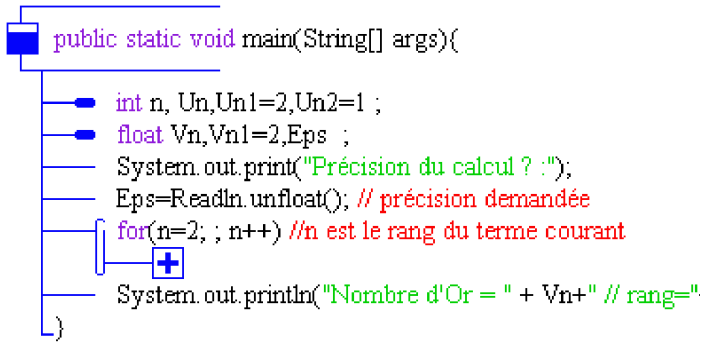
Implantation en Java

On entre au clavier un nombre réel ci-dessous 0.00001, pour la précision choisie (ici 5 chiffres après la virgule), puis le programme calcule et affiche le Nombre d'or (les caractères gras représentent ce qui est écrit par le programme, les italiques ce qui est entré au clavier) :

Précision du calcul ? : *0.00001*
Nombre d'Or = 1.6180328 // rang=14

Proposition de squelette de classe Java à implanter avec un boucle **for :**

class AppliNombredOr {



}

Remarquons que nous proposons une boucle **for** ne contenant pas de condition de rebouclage dans son en-tête (donc en apparence infinie), puisque nous effectuerons le test "**si** |Vn - Vn1| <= Eps **alors** Arrêt de la boucle" qui permet l'arrêt de la boucle. Dans cette éventualité , la boucle **for** devra donc contenir dans son corps, une instruction de rupture de séquence.

Algorithme
Conjecture de Goldbach

Objectif : On souhaite écrire un programme Java afin de vérifier sur des exemples, la conjecture de GoldBach (1742), soit : "Tout nombre pair est décomposable en la somme de deux nombres premiers".

Dans cet exercice nous réutilisons un algorithme déjà traité (algorithme du test de la primalité d'un nombre entier), nous rappelons ci-après un algorithme indiquant si un entier "nbr" est premier ou non :

```
Algorithme Premier
  Entrée: nbr  N
  Local: Est_premier  {Vrai , Faux}
         divis,compt  N² ;

début
 lire(nbr);
 divis   3;
 Est_premier   Vrai;
 Répéter
   Si reste(nbr par divis) = 0 Alors
    Est_premier   Faux
   Sinon
    divis   divis+2
   Fsi
 jusquà (divis > nbr / 2)ou (Est_premier=Faux);
 Si Est_premier = Vrai Alors
   ecrire(nbr est premier)
 Sinon
   ecrire(nbr n'est pas premier)
 Fsi
FinPremier
```

Conseil :

Il faudra traduire cet algorithme en fonction recevant comme paramètre d'entrée le nombre entier dont on teste la primalité, et renvoyant un booléen **true** ou **false** selon que le nombre entré a été trouvé ou non premier

Spécifications de l'algorithme de Goldbach :

En deux étapes :

1. On entre un nombre pair n au clavier, puis on génère tous les couples (a,b) tels que a + b = n, en faisant varier a de 1 à n/2. Si l'on rencontre un couple tel que a et b soient simultanément premiers la conjecture est vérifiée.

2. On peut alors, au choix soit arrêter le programme, soit continuer la recherche sur un autre nombre pair.

Exemple :
Pour n = 10, on génère les couples :
(1,9), (2,8), (3,7), (4,6), (5,5)
on constate que la conjecture est vérifiée, et on écrit :
10 = 3 + 7
10 = 5 + 5

Implantation en Java

On écrira la méthode booléenne **EstPremier** pour déterminer si un nombre est premier ou non, et la méthode **generCouples** qui génère les couples répondant à la conjecture.

Proposition de squelette de classe Java à implanter :

```
class ApplicationGoldBach  {
    public static void main(String[ ] args) {
       ……..
    }
    static boolean EstPremier(int m)  {
       ……..
    }
    static void generCouples(int n)  {
       ……..
    }
 }
```

Algorithme
Méthodes d'opérations sur 8 bits

Objectif : On souhaite écrire une application de manipulation interne des bits d'une variable entière non signée sur 8 bits. Le but de l'exercice est de construire une famille de méthodes de travail sur un ou plusieurs bits d'une mémoire. L'application à construire contiendra 9 méthodes :

Spécifications des méthodes :

1. Une méthode **BitSET** permettant de mettre à **1** un bit de rang fixé.
2. Une méthode **BitCLR** permettant de mettre à **0** un bit de rang fixé.
3. Une méthode **BitCHG** permettant de remplacer un bit de rang fixé par son complément.
4. Une méthode **SetValBit** permettant de modifier un bit de rang fixé.
5. Une méthode **DecalageD** permettant de décaler les bits d'un entier, sur la droite de **n** positions (introduction de **n** zéros à gauche).
6. Une méthode **DecalageG** permettant de décaler les bits d'un entier, sur la gauche de **n** positions (introduction de **n** zéros à droite).
7. Une méthode **BitRang** renvoyant le bit de rang fixé d'un entier.
8. Une méthode **ROL** permettant de décaler avec rotation, les bits d'un entier, sur la droite de **n** positions (réintroduction à gauche).
9. Une méthode **ROR** permettant de décaler avec rotation, les bits d'un entier, sur la gauche de **n** positions (réintroduction à droite).

Exemples de résultats attendus :

Prenons une variable X entier (par exemple sur 8 bits) X = **11100010**

1. **BitSET** (X,3) = X = **11101010**
2. **BitCLR** (X,6) = X = **10100010**
3. **BitCHG** (X,3) = X = **11101010**
4. **SetValBit**(X,3,1) = X = **11101010**
5. **DecalageD** (X,3) = X = **00011100**
6. **DecalageG** (X,3) = X = **00010000**
7. **BitRang** (X,3) = 0 ; BitRang (X,6) = 1 ...
8. **ROL** (X,3) = X = **00010111**
9. **ROR** (X,3) = X = **01011100**

Implantation en Java

La conception de ces méthodes ne dépendant pas du nombre de bits de la mémoire à opérer on choisira le type int comme base pour une mémoire. Ces méthodes sont classiquement des outils de manipulation de l'information au niveau du bit.

Lors des jeux de tests pour des raisons de simplicité de lecture il est conseillé de ne rentrer que des valeurs entières portant sur 8 bits.

Il est bien de se rappeler que le type primaire **int** est un type entier signé sur 32 bits (représentation en complément à deux).

Proposition de squelette de classe Java à implanter :

```
class Application8Bits {
    public static void main(String [ ] args){
        ……..       }
    static int BitSET (int nbr, int num) {  ……..      }
    static int BitCLR (int nbr, int num) {  ……..      }
    static int BitCHG (int nbr, int num) {  ……..      }
    static int SetValBit (int nbr, int rang, int val) {  ……..      }
    static int DecalageD (int nbr, int n) {  ……..      }
    static int DecalageG (int nbr, int n) {  ……..     }
    static int BitRang (int nbr, int rang) {  ……..   }
    static int ROL (int nbr, int n) {   ……..    }
    static int ROR (int nbr, int n) {  ……..    }
  }
```

SOLUTIONS DES ALGORITHMES EN JAVA

Classe Java solution

Calcul de la valeur absolue d'un nombre réel

Objectif : Ecrire un programme Java servant à calculer la valeur absolue d'un nombre réel x à partir de la définition de la valeur absolue. La valeur absolue du nombre réel x est le nombre réel |x| :

|x| = x , si x □□

|x| = -x si x < 0

Spécifications de l'algorithme :

lire(x);
si x>=0 **alors** écrire('|x| =', x)
sinon écrire('|x| =', -x)
fsi

Implantation en Java avec un if…else :

```
class ApplicationValAbsolue
{
   static void main(String[ ] args)  {
      float x;
      System.out.print("Entrez un nombre x = ");
      x = Readln.unfloat();
      if (x<0)
         System.out.println("|x| = "+(-x));
      else
         System.out.println("|x| = "+x);
   }
}
```

Implantation en Java avec un "... ? ... : ..." :

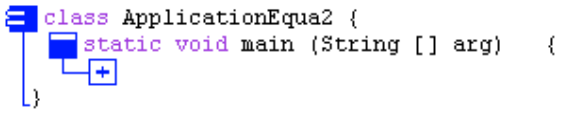
```
class ApplicationValAbsolue {
   static void main(String[ ] args) {
      float x;
      System.out.print("Entrez un nombre x = ");
      x = Readln.unfloat();
      System.out.println("|x| = "+ (x<0 ? -x : x) );
   }
}
```

Classe Java solution

Algorithme de résolution de l'équation du second degré dans R.

Objectif : Ecrire un programme Java de résolution dans R de l'équation du second degré : $Ax^2 + Bx + C = 0$

Implantation en Java de la classe



```
class ApplicationEqua2 {
    static void main (String [] arg)    {
    }
}
```

Implantation en Java de la méthode main :

```
static void main (String [] arg)    {
    double a, b, c, delta ;
    double x, x1, x2 ;
    System.out.print("Entrer une valeur pour a : ") ;
    a =  Readln.undouble() ;
    System.out.print("Entrer une valeur pour b : ") ;
    b =  Readln.undouble() ;
    System.out.print("Entrer une valeur pour c : ") ;
    c =  Readln.undouble() ;
    if (a ==0)  {
        if (b ==0)  {
            if (c ==0)    {
                System.out.println("tout reel est solution") ;
            }
            else  {
                System.out.println("il n'y a pas de solution") ;
            }
        }
        else  {
            x = -c/b ;
            System.out.println("la solution est " + x) ;
        }
    }
    else  {
        delta = b*b -4*a*c ;
        if (delta <0)    {
            System.out.println("il n'y a pas de solution dans les reels") ;
        }
        else  {
            x1 = (-b + Math.sqrt(delta))/ (2*a) ;
            x2 = (-b - Math.sqrt(delta))/ (2*a) ;
            System.out.println("il y deux solutions egales a " + x1 + " et " + x2) ;
        }
    }
}
```

Classe Java solution

Calcul des nombres de Armstrong

Objectif : On dénomme nombre de Armstrong un entier naturel qui est égal à la somme des cubes des chiffres qui le composent. Ecrire un programme Java qui affiche de tels nombres.

***Exemple* :**

$153 = 1^3 + 5^3 + 3^3$

153 = 1 + 125 + 27, est un nombre de Armstrong.

Implantation en Java

```
class ApplicationArmstrong {
   static void main(String[ ] args) {
      int i, j, k, n, somcube;
      System.out.println("Nombres de Armstrong:");
      for(i = 1; i<=9; i++)
         for(j = 0; j<=9; j++)
            for(k = 0; k<=9; k++)
            { n = 100*i + 10*j + k;
              somcube = i*i*i + j*j*j + k*k*k;
              if (somcube == n) System.out.println(n);
            }
   }
}
```

Classe Java solution

Calcul de nombres parfaits

Objectif : On souhaite écrire un programme java de calcul des n premiers nombres parfaits. Un nombre est dit parfait s'il est égal à la somme de ses diviseurs, 1 compris. Ecrire un programme Java qui affiche de tels nombres.

***Exemple* : 6 = 1+2+3 , est un nombre parfait.**

Implantation en Java

Ci-dessous le programme Java complet qui produit le dialogue suivant à l'écran (les caractères gras représentent ce qui est écrit par le programme, les italiques ce qui est entré au clavier) :

Entrez combien de nombre parfaits : *4*
6 est un nombre parfait
28 est un nombre parfait
496 est un nombre parfait
8128 est un nombre parfait

```
class ApplicationParfaits {
   static void main(String[ ] args) {
      int compt = 0, n, k, somdiv, nbr;
      System.out.print("Entrez combien de nombre parfaits : ");
      n = Readln.unint( );
      nbr = 2;
      while (compt != n)
      { somdiv = 1;
        k = 2;
        while(k <= nbr/2 )
        {
          if (nbr % k == 0) somdiv += k ;
          k++;
        }
        if (somdiv == nbr)
        { System.out.println(nbr+" est un nombre parfait");
          compt++;
        }
        nbr++;
      }
   }
}
```

Classe Java solution

Calcul du pgcd de 2 entiers (méthode Euclide)

Objectif : Ecrire un programme de calcul du pgcd de deux entiers non nuls, en Java à partir de l'algorithme de la méthode d'Euclide.

Implantation en Java

Ci-dessous le programme Java complet qui produit le dialogue suivant à l'écran (les caractères gras représentent ce qui est écrit par le programme, les italiques ce qui est entré au clavier) :

Entrez le premier nombre : *21*
Entrez le deuxième nombre : *45*
Le PGCD de 21 et 45 est : 3

```
                    // PGCD - EUCLIDE  avec des méthodes
class TD2ApplicationPgcdEuclide{

   static int Pgcd(int a, int b)
   { // renvoie le pgcd de a>=b
      int r;
      do {
         r = a % b;
         a = b;
         b = r;
      }while(r !=0);
      return a;
   }

   static int Max(int a, int b)
   { // renvoie le max de a et de b
      return a>b ? a : b;
   }

   static int Min(int a, int b)
   { // renvoie le min de a et de b
      return a>b ? b : a;
   }

   public static void main(String[ ] args)
   {
      int a = Readln.unint();
      int b = Readln.unint();
      int max = Max(a,b), min = Min(a,b);
      a = max;
      b = min;
      if(min !=0)
         System.out.println("pgcd de "+a+" et de "+b+" = "+Pgcd(a,b));
      else System.out.println("calcul impossible");
   }
}
```

Classe Java solution

Calcul du pgcd de 2 entiers (méthode Egyptienne)

Objectif : Ecrire un programme de calcul du pgcd de deux entiers non nuls, en Java à partir de l'algorithme de la méthode dite "égyptienne ".

Implantation en Java

Ci-dessous le programme Java complet qui produit le dialogue suivant à l'écran (les caractères gras représentent ce qui est écrit par le programme, les italiques ce qui est entré au clavier) :

Entrez le premier nombre : *21*
Entrez le deuxième nombre : *45*
Le PGCD de 21 et 45 est : 3

```
class ApplicationEgyptien {

    static void main(String[ ] args) {
        int p,q;
        System.out.print("Entrez le premier nombre : ");
        p = Readln.unint( );// méthode permettant de lire un entier au clavier
        System.out.print("Entrez le deuxième nombre : ");
        q = Readln.unint( );
        if (p*q!=0)
            System.out.println("Le pgcd de "+p+" et de "+q+" est "+pgcd(p,q));
        else
            System.out.println("Le pgcd n'existe pas lorsque l'un des deux nombres est nul !");
    }


    static int pgcd (int p, int q) {
        while ( p != q) {
            if (p>q) p -= q;
            else q -= p;
        }
        return p;
    }

}
```

Classe Java solution

Calcul de nombres premiers (boucles **while** et **do**…**while**)

Objectif : On souhaite écrire un programme Java de calcul et d'affichage des n premiers nombres premiers.

Implantation en Java avec une boucle while et une boucle do...while imbriquée

```
                //une liste des n premiers nombres premiers
class ApplicationComptPremiers1
{
  static public void main(String[ ] args)
  {
    int divis, nbr, n, compt = 0 ;
    boolean Est_premier;
    System.out.print("Combien de nombres premiers :  ");
    n = Readln.unint();
    System.out.println( 2 );
    nbr=3;
    while (compt < n-1) {
      divis=2 ;
      Est_premier=true;
      do{
        if(nbr % divis ==0)   Est_premier=false;
        else divis = divis+1 ;
      }while ((divis <= nbr/2)&& (Est_premier==true));
      if(Est_premier) {
        compt++;
        System.out.println( nbr );
       }
      nbr++ ;
    }
  }
}
```

Ce programme Java produit le dialogue suivant à l'écran:

```
Combien de nombres premiers :  5
  2
  3
  5
  7
  11
```

Classe Java solution

Calcul de nombres premiers (boucles **for**)

Objectif : On souhaite écrire un programme Java de calcul et d'affichage des n premiers nombres premiers.

Implantation en Java avec deux boucles for imbriquées :

```
class ApplicationComptPremiers2 {

   static void main(String[ ] args) {
      int divis, nbr, n, compt = 1 ;
      boolean Est_premier;
      System.out.print("Combien de nombres premiers :  ");
      n = Readln.unint();
      System.out.println( 2 );
      for( nbr = 3; compt < n  ; nbr += 2 )
      { Est_premier = true;
        for (divis = 2; divis<= nbr/2; divis++ )
           if ( nbr % divis == 0 )
           { Est_premier = false;
             break;
           }
        if (Est_premier)
        {
           compt++;
           System.out.println( nbr );
        }
      }
   }
}
```

Le programme Java précédent produit le dialogue suivant à l'écran :

Combien de nombres premiers : *5*
2
3
5
7
11

Classe Java solution
Calcul du nombre d'or

Objectif : On souhaite écrire un programme Java qui calcule le nombre d'or utilisé par les anciens comme nombre idéal pour la sculpture et l'architecture (nbr d'Or = 1,6180339887498948482045868343656 4).

Implantation en Java avec un boucle for :

Le programme ci-dessous donne une valeur approchée avec une précision fixée de □du nombre d'or. Le programme indique en outre le rang du dernier terme de la suite correspondant à la valeur approchée calculée.

Exemple : On entre au clavier un nombre réel ci-dessous 0.00001, pour la précision choisie (ici 5 chiffres après la virgule), puis le programme calcule et affiche le Nombre d'or (les caractères gras représentent ce qui est écrit par le programme, les italiques ce qui est entré au clavier) :

Précision du calcul ? : *0.00001*
Nombre d'Or = 1.6180328 // rang=14

```
class AppliNombredOr {

  static void main(String[ ] args) {
    int n, Un, Un1=2, Un2=1 ;
    float Vn,Vn1=2, Eps  ;
    System.out.print("Précision du calcul ? :");
    Eps=Readln.unfloat(); // précision demandée
    for (n=2; ; n++) //n est le rang du terme courant
    {
      Un = Un1 + Un2;
      Vn =(float)Un / (float)Un1;
      if (Math.abs(Vn - Vn1) <= Eps)
        break;
      else
      {
        Un2 = Un1;
        Un1 = Un;
        Vn1 = Vn;
      }
    }
    System.out.println("Nombre d'Or = " + Vn+" // rang="+n);
  }
}
```

Classe Java solution
Conjecture de Goldbach

Objectif : On souhaite écrire un programme Java afin de vérifier sur des exemples, la conjecture de GoldBach (1742), soit : "Tout nombre pair est décomposable en la somme de deux nombres premiers".

Implantation en Java :

```
class ApplicationGoldBach {

   static void main(String[ ] args) {
      int n;
      while ( (n=Readln.unint( )) !=0 ){
         generCouples(n); }
   }

   static boolean EstPremier(int m)  {
      int k ;
      for (k = 2 ; k <= m / 2 ; k++) {
         if (m % k == 0) {
            return  false;
         }
      }
      return true;
   }

   static void generCouples(int n) {
      if (n % 2 ==0) {
         for (int a = 1; a <= n/2; a++) {
            int  b;
            b = n - a;
            if ( EstPremier(a) && EstPremier(b) ) {
               System.out.println(n+" = "+a+" + "+b);
            }
         }
      }
      else System.out.println("Votre nombre n'est pas pair !");
   }
}
```

Classe Java solution
Méthodes d'opérations sur 8 bits

Objectif : On souhaite écrire une application de manipulation interne des bits d'une variable entière non signée sur 8 bits.

Implantation en Java

La conception de ces méthodes ne dépendant pas du nombre de bits de la mémoire à opérer on choisira le type int comme base pour une mémoire. Ces méthodes sont classiquement des outils de manipulation de l'information au niveau du bit.

Lors des jeux de tests pour des raisons de simplicité de lecture il est conseillé de ne rentrer que des valeurs entières portant sur 8 bits.

La classe Application8Bits en général :

```
class Application8Bits {
//  Integer.MAX_VALUE : 32  bit = 2147483647
//  long.MAX_VALUE : 64 bits = 9223372036854775808

  static void main(String[] args){

  static  int  BitSET(int nbr, int num)

  static  int  BitCLR(int nbr, int num)

  static int  BitCHG(int nbr, int num)

  static int DecalageD (int nbr, int n)

  static int DecalageG (int nbr, int n)

  static int BitRang (int nbr, int rang)

  static int  SetValBit (int nbr, int rang,int val)

  static int ROL (int nbr, int n)

  static int ROR (int nbr,int n)
}
```

La méthode main de la classe Application8Bits :

```
static void main(String[] args){
  int n,p,q,r,t;
  n =9;// 000...1001
  System.out.println("n=9 : n="+Integer.toBinaryString(n));
  p = BitCLR(n,3);// p=1
  System.out.println("BitCLR(n,3) ="+Integer.toBinaryString(p));
  q = BitSET(n,2);// q=13
  System.out.println("BitSET(n,2) ="+Integer.toBinaryString(q));
  r = BitCHG(n,3);// r=1
  System.out.println("BitCHG(n,3) ="+Integer.toBinaryString(r));
  t = BitCHG(n,2);// t=13
  System.out.println("BitCHG(n,2) ="+Integer.toBinaryString(t));
  System.out.println("p = "+p+", q = "+q+", r = "+r+", t = "+t);
  n =-2147483648;//1000.....00 entier minimal
  System.out.println("n=-2^31 : n="+Integer.toBinaryString(n));
  p=ROL(n,3);// 000...000100 => p=4
  System.out.println("p = "+p);
  n =-2147483648+1;//1000.....01 entier minimal+1
  System.out.println("n=-2^31+1 : n="+Integer.toBinaryString(n));
  p=ROL(n,3);// 000...0001100 => p=12
  System.out.println("p = "+p);
  n =3;//0000.....0 11
  System.out.println("n=3 : n="+Integer.toBinaryString(n));
  p=ROR(n,1);//100000...001 => p=-2147483647
  System.out.println("ROR(n,1) = "+p+"= "+Integer.toBinaryString(p));
  p=ROR(n,2);// 11000...000 => p= -1073741824
  System.out.println("ROR(n,2) = "+p+"= "+Integer.toBinaryString(p));
  p=ROR(n,3);// 011000...000 => p= +1610612736 =2^30+2^29
  System.out.println("ROR(n,3) = "+p+"= "+Integer.toBinaryString(p));
}
```

Les autres méthodes de la classe Application8Bits :

```
static  int  BitSET(int nbr, int num)
{ // positionne à 1 le bit de rang num
  int mask;
  mask =1<< num;
  return nbr | mask;
}

static  int  BitCLR(int nbr, int num)
{ // positionne à 0 le bit de rang num
  int mask;
  mask = ~ (1<< num);
  return nbr & mask;
}

static int  BitCHG(int nbr, int num)
{ // complémente le bit de rang num (0 si bit=1, 1 si bit=0)
  int mask;
  mask =1<< num;
  return nbr ^ mask;
}
```

Les autres méthodes de la classe Application8Bits (suite) :

```
static int DecalageD (int nbr, int n)
{ // décalage sans le signe de n bits vers la droite
  return nbr >>> n ;
}

static int DecalageG (int nbr, int n)
{ // décalage de 2 bits vers la gauche
  return nbr << n ;
}

static int BitRang (int nbr, int rang)
{ //renvoie le bit de rang fixé
  return (nbr >>> rang ) % 2;
}

static int  SetValBit (int nbr, int rang,int val)
{ //positionne à val le bit de rang fixé
  return    val ==0 ? BitCLR( nbr,rang):BitSET( nbr,rang) ;
}

static int ROL (int nbr, int n)
{ //décalage à gauche avec rotation
  int C;
  int N = nbr;
  for(int i=1; i<=n; i++)
  {
    C = BitRang(N,31);
    N = N <<1;
    N = SetValBit(N,0,C);
  }
  return N ;
}

static int ROR (int nbr,int n)
{ //décalage à droite avec rotation
  int C;
  int N = nbr;
  for(int i=1; i<=n; i++)
  {
    C = BitRang(N,0);
    N = N >>> 1;
    N = SetValBit(N,31,C);
  }
  return N ;
}
```

Résultats produits par la méthode main :

Nous avons utilisé `Integer.toBinaryString(n)` qui est la méthode de conversion de la classe Integer permettant d'obtenir sous forme d'une chaîne le contenu transformé en binaire d'une mémoire de type int. Cette méthode nous permet d'ausculter le contenu d'une mémoire en binaire afin de voir ce qui s'est passé lors de l'utilisation d'une des méthodes précédentes.

Le programme Java précédent produit les lignes suivantes :

n=9 : n=1001
BitCLR(n,3) =1
BitSET(n,2) =1101

```
BitCHG(n,3) =1
BitCHG(n,2) =1101
p = 1, q = 13, r = 1, t = 13
n=-2^31 : n=10000000000000000000000000000000
p = 4
n=-2^31+1 : n=10000000000000000000000000000001
p = 12
n=3 : n=11
ROR(n,1) = -2147483647= 10000000000000000000000000000001
ROR(n,2) = -1073741824= 11000000000000000000000000000000
ROR(n,3) = 1610612736= 01100000000000000000000000000000
```

Méthodes de traitement de chaînes

Objectif : Soit à implémenter sous forme de méthodes Java d'une classe, les six procédures ou fonctions spécifiées ci-dessous en Delphi.

function Length(S:string): Integer;

La fonction Length renvoie le nombre de caractères effectivement utilisés dans la chaîne ou le nombre d'éléments dans le tableau.

function Concat(s1, s2 : string): string;

Utilisez Concat afin de concaténer deux chaînes. Chaque paramètre est une expression de type chaîne. Le résultat est la concaténation des deux paramètres chaîne.

L'utilisation de l'opérateur plus (+) sur deux chaînes a le même effet que l'utilisation de la fonction Concat :

S := 'ABC' + 'DEF';

procedure Delete(**var** S: string; Index, Count:Integer);

La procédure Delete supprime une sous-chaîne de Count caractères dans la chaîne qui débute à la position S[Index]. S est une variable de type chaîne. Index et Count sont des expressions de type entier.

Si Index est plus grand que la taille de S, aucun caractère n'est supprimé. Si Count spécifie un nombre de caractères supérieur à ceux qui restent en partant de S[Index], Delete supprime tous les caractères jusqu'à la fin de la chaîne.

procedure Insert(Source: string; **var** S: string; Index: Integer);

Insert fusionne la chaîne Source dans S à la position S[index].

function Copy(S; Index, Count: Integer): string;

S est une expression du type chaîne. Index et Count sont des expressions de type entier. Copy renvoie une sous-chaîne ou un sous-tableau contenant Count caractères ou éléments en partant de S[Index].

function Pos (Substr: string; S: string): Integer;

La fonction Pos recherche une sous-chaîne, Substr, à l'intérieur d'une chaîne. Substr et S sont des expressions de type chaîne. Pos recherche Substr à l'intérieur de S et renvoie une valeur entière correspondant à l'indice du premier caractère de Substr à l'intérieur de S. Pos fait la distinction majuscules/minuscules. Si Substr est introuvable, Pos renvoie zéro.

Proposition de squelette de classes Java à implanter :

*La classe string contenant les méthodes de traitement des **String** :*

```
class string {

    static String delete(String s,int debut, int fin) {

    static String insert(String Source, String s, int index) {

    static int length(String s) {

    static String concat(String s1, String s2) {

    static String concat(String s1, String s2, String s3) {

    static String concat(String s1, String s2, String s3, String s4) {

    static String copy(String s, int index, int count) {

    static int pos(String substr, String s) {

}
```

La classe principale contenant la méthode main et utilisant les méthodes de la classe string :

```
class testExostring {
  //--> il n'y a pas de confusion possible entre string et String !

  public static void main(String[] Args) {
    String s = "abcdefghijklmn";
    System.out.println("s = "+s);
    System.out.println("length(s) = "+string.length(s));
    System.out.println("copy(s,2,6) = "+string.copy(s,2,6));
    System.out.println("delete(s,2,6) = "+string.delete(s,2,6));
    System.out.println("insert('xyz',s,2) = "+string.insert("xyz",s,2));
    System.out.println("pos('efghi',s) = "+string.pos("efghi",s));
    System.out.println("pos('efghk',s) = "+string.pos("efghk",s));
  }
}
```

Solution

Méthodes de traitement de chaînes

La classe string utilisant des méthodes de la classe **String** :

```
class string {

   static String delete(String s,int debut, int fin) {
      String t1 = s.substring(0,debut-1);
      String t2 = s.substring(fin,s.length());
      return   t1+t2;
   }

   static String insert(String Source, String s, int index) {
      String t1 = s.substring(0,index-1);
      String t2 = s.substring(index-1,s.length());
      return   t1+Source+t2;
   }

   static int length(String s) {
      return  s.length();
   }

   static String concat(String s1, String s2) {
      return   s1+s2;
   }

   static String copy(String s, int index, int count) {
      return  s.substring(index-1,index+count-2);
   }

   static int pos(String substr, String s) {
      return s.indexOf(substr)+1;
   }

}
```

La classe principale contenant la méthode main et utilisant les méthodes de la classe string :

```
class testExostring {
   //--> il n'y a pas de confusion possible entre string et String !

   public static void main(String[] Args) {
      String s = "abcdefghijklmn";
      System.out.println("s = "+s);
      System.out.println("length(s) = "+string.length(s));
      System.out.println("copy(s,2,6) = "+string.copy(s,2,6));
      System.out.println("delete(s,2,6) = "+string.delete(s,2,6));
      System.out.println("insert('xyz',s,2) = "+string.insert("xyz",s,2));
      System.out.println("pos('efghi',s) = "+string.pos("efghi",s));
      System.out.println("pos('efghk',s) = "+string.pos("efghk",s));
   }
}
```

TD String Java
phrase palindrome (première version)

Voici le squelette du programme Java à écrire :

```
/*
  phrases palindromes :
     et la marine s'en ira, malte.
     esope reste ici et se repose.
     elu par cette crapule.
*/

class palindromel {

   static String compresser(String s){
   }

   static String Inverser(String s){
   }

   public static void main(String[] x){

      System.out.print("Entrez une phrase : ");
      String phrase=Readln.unstring();
      String strMot=compresser(phrase);
      System.out.println("strMot="+strMot);
      String strInv=Inverser(strMot);
      System.out.println("strInv="+strInv);

      if(strMot.equals(strInv))
         System.out.println("palindrome");
      else System.out.println("non palindrome");
   }
}
```

Travail à effectuer :

Ecrire les méthode **compresser** et **Inverser** , il est demandé d'écrire une première version de la méthode **Inverser**.

- La première version de la méthode **Inverser** construira une chaîne locale à la méthode caractère par caractère avec une boucle **for** à un seul indice.

Solutions String Java
phrase palindrome (première version)

La méthode **compresser** :

```
static String compresser(String s){
  String strLoc="";
  for(int i=0; i<s.length(); i++){
    if(s.charAt(i) !=' ' && s.charAt(i) !=','
        && s.charAt(i) !='\'' && s.charAt(i) !='.')
      strLoc +=s.charAt(i);
  }
  return strLoc;
}
```

La méthode **compresser** élimine les caractères non recevables comme : blanc, virgule, point et apostrophe de la **String** s passée en paramètre.

Remarquons que l'instruction strLoc +=s.charAt(i) permet de concaténer les caractères recevables de la chaîne locale strLoc, par balayage de la String s depuis le caractère de rang 0 jusqu'au caractère de rang s.length()-1.

La référence de String strLoc pointe à chaque tour de la boucle **for** vers un nouvel objet créé par l'opérateur de concaténation +

La première version de la méthode **Inverser** :

```
static String Inverser(String s){
  String strLoc="";
  for(int i=0; i<s.length(); i++)
    strLoc =s.charAt(i)+strLoc;
  return strLoc;
}
```

TD String-tableau
phrase palindrome (deuxième version)

Voici le squelette du programme Java à écrire :

```
class palindromel {

  static String compresser(String s){
  }

  static String Inverser(String s){
  }

  public static void main(String[] x){
    System.out.print("Entrez une phrase : ");
    String phrase=Readln.unstring();
    String strMot=compresser(phrase);
    System.out.println("strMot="+strMot);
    String strInv=Inverser(strMot);
    System.out.println("strInv="+strInv);

    if(strMot.equals(strInv))
      System.out.println("palindrome");
    else System.out.println("non palindrome");
  }
}
```

Travail à effectuer :

Ecrire les méthode **compresser** et **Inverser** , il est demandé d'écrier une deuxième version de la méthode **Inverser**.

- La deuxième version de la méthode **Inverser** modifiera les positions des caractères ayant des positions symétriques dans la chaîne avec une boucle **for** à deux indices et en utilisant un tableau de **char**.

La méthode **compresser** a déjà été développée auparavant et reste la même :

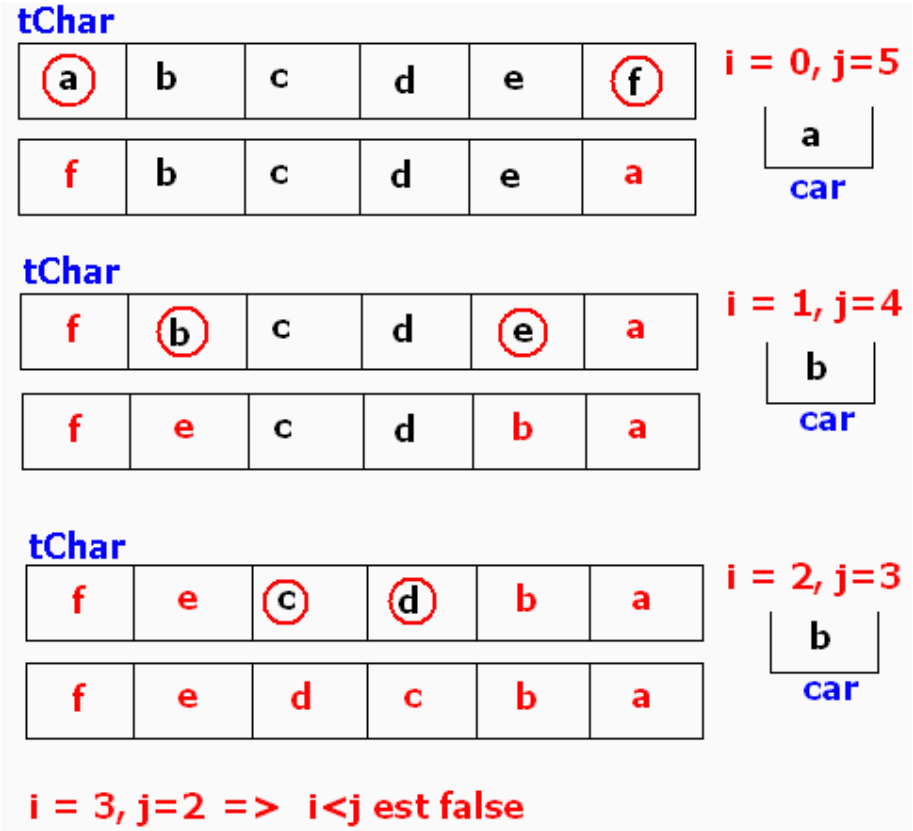
```
static String compresser(String s){
  String strLoc="";
  for(int i=0; i<s.length(); i++){
    if(s.charAt(i) !=' ' && s.charAt(i) !=','
        && s.charAt(i) !='\'' && s.charAt(i) !='.')
      strLoc +=s.charAt(i);
  }
  return strLoc;
}
```

Solution String-tableau
phrase palindrome (deuxième version)

La deuxième version de la méthode **Inverser** :

```
static String Inverser(String s){
  char [ ] tChar =s.toCharArray();
  char car ;
  for ( int i=0 , j=tChar.length-1 ; i<j; i++, j--){
    car = tChar[i];
    tChar[i ]= tChar[j];
    tChar[j] = car;
  }
  return new String(tChar);
}
```

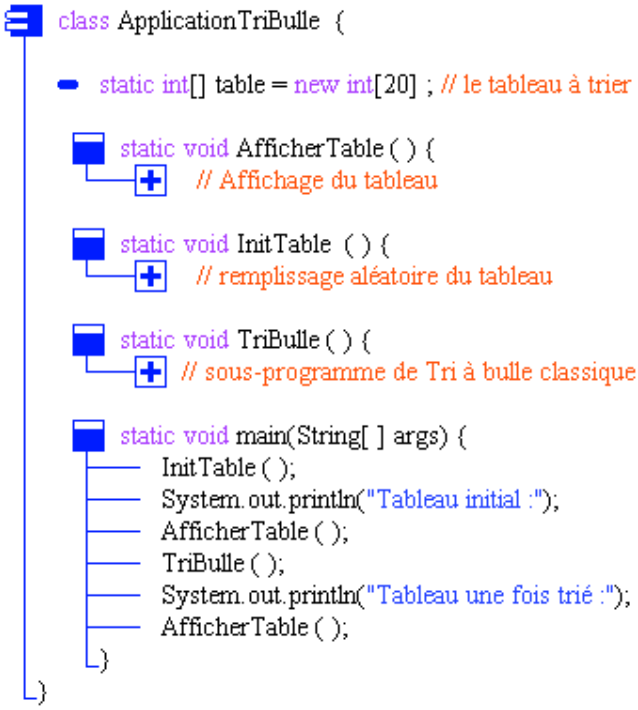
Trace d'exécution sur la chaîne s = "abcdef" :



Algorithme
Tri à bulles sur un tableau d'entiers

Objectif : Ecrire un programme Java implémentant l'algorithme du tri à bulles.

Proposition de squelette de classe Java à implanter :



```
class ApplicationTriBulle {

    static int[] table = new int[20] ; // le tableau à trier

    static void AfficherTable ( ) {
        // Affichage du tableau

    static void InitTable ( ) {
        // remplissage aléatoire du tableau

    static void TriBulle ( ) {
        // sous-programme de Tri à bulle classique :

    static void main(String[ ] args) {
        InitTable ( );
        System.out.println("Tableau initial :");
        AfficherTable ( );
        TriBulle ( );
        System.out.println("Tableau une fois trié :");
        AfficherTable ( );
    }
}
```

Spécifications de l'algorithme :

Algorithme Tri_a_Bulles
local: i , j , **n**, temp □ Entiers naturels
Entrée - Sortie : Tab □ Tableau d'Entiers naturels de **1** à **n** éléments
début
pour i **de** **n** **jusqu**à 1 **faire** *// recommence une sous-suite (**a**1, **a**2, ... , **a**i)*
pour j **de** 2 **jusqu**à i **faire** *// échange des couples non classés de la sous-suite*
si Tab[j-1] > Tab[j] **alors** *// **a**j-1et **a**j non ordonnés*
temp □ Tab[j-1] ;
Tab[j-1] □ Tab[j] ;
Tab[j] □ temp *//on échange les positions de **a**j-1et **a**j*
Fsi
fpour
fpour
Fin Tri_a_Bulles

Solution en Java
Tri à bulles sur un tableau d'entiers

La méthode Java implantant l'algorithme de tri à bulle :

```
static void TriBulle ( ) {
/* sous-programme de Tri à bulle classique :
   on trie les éléments du n°1 au  n°19   */
 int n = table.length-1;
 for ( int i = n; i>=1; i--)
    for ( int j = 2; j <= i; j++)
       if (table[j-1] > table[j]){
          int temp = table[j-1];
          table[j-1] = table[j];
          table[j] = temp;
       }
}
```

Une classe contenant cette méthode et la testant :

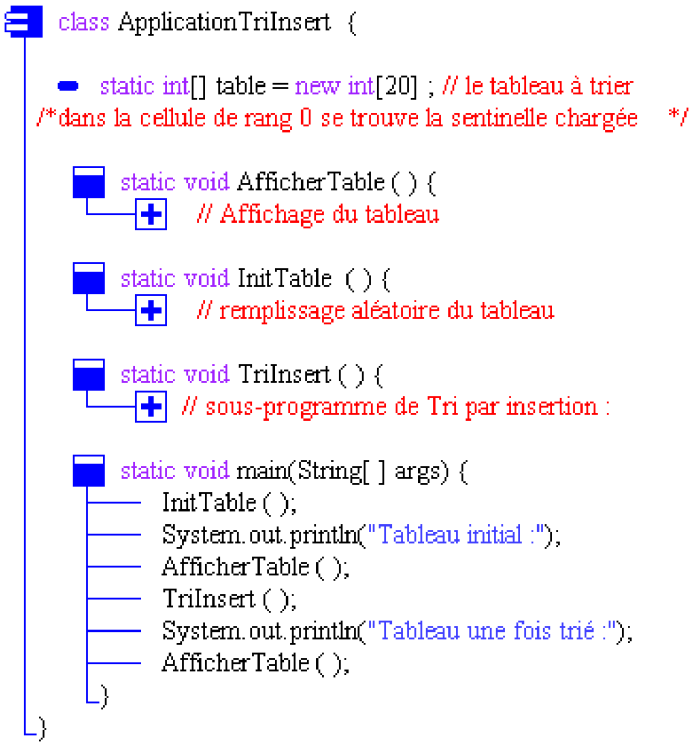
```
class ApplicationTriBulle  {
 //-- le tableau à trier : 20 éléments
 static int[] table = new int[20] ;

 static void TriBulle ( ) {
    /* sous-programme de Tri à bulle classique :

 static void AfficherTable ( ) {
 //-- Affichage du tableau
    int n = table.length-1;
    for ( int i = 1; i <= n; i++)
       System.out.print(table[i]+" , ");
    System.out.println();
 }

 static void InitTable  ( ) {
 //-- remplissage aléatoire du tableau
    int n = table.length-1;
    for ( int i = 1; i <= n; i++)
       table[i] = (int)(Math.random()*100);
 }

 public static void main(String[ ] args) {
    InitTable ( );
    System.out.println("Tableau initial :");
    AfficherTable ( );
    TriBulle ( );
    System.out.println("Tableau une fois trié :");
    AfficherTable ( );
 }
}
```

Tableau initial :
3 , 97 , 27 , 2 , 56 , 67 , 25 , 87 , 41 , 2 , 80 , 73 , 61 , 97 , 46 , 92 , 38 , 70 , 32 ,
Tableau une fois trié :
2 , 2 , 3 , 25 , 27 , 32 , 38 , 41 , 46 , 56 , 61 , 67 , 70 , 73 , 80 , 87 , 92 , 97 , 97 ,

Algorithme
Tri par insertion sur un tableau d'entiers

Objectif : Ecrire un programme Java implémentant l'algorithme du tri par insertion.

Proposition de squelette de classe Java à implanter :



```
class ApplicationTriInsert  {

   static int[] table = new int[20] ; // le tableau à trier
/*dans la cellule de rang 0 se trouve la sentinelle chargée    */

    static void AfficherTable ( ) {
        // Affichage du tableau

    static void InitTable  ( ) {
        // remplissage aléatoire du tableau

    static void TriInsert ( ) {
        // sous-programme de Tri par insertion :

    static void main(String[ ] args) {
        InitTable ( );
        System.out.println("Tableau initial :");
        AfficherTable ( );
        TriInsert ( );
        System.out.println("Tableau une fois trié :");
        AfficherTable ( );
    }
}
```

Spécifications de l'algorithme :

Algorithme Tri_Insertion
local: i , j , **n**, v □ Entiers naturels
Entrée : Tab □ Tableau d'Entiers naturels de 0 à **n** éléments
Sortie : Tab □ Tableau d'Entiers naturels de 0 à **n** éléments (le même tableau)

{ *dans la cellule de rang 0 se trouve une sentinelle chargée d'éviter de tester dans la boucle* **tantque** .. **faire** *si l'indice j n'est pas inférieur à 1, elle aura une valeur inférieure à toute valeur possible de la liste*
}

```
début


 pour i de2 jusquà n faire// la partie non encore triée (ai, ai+1, ... , an)
    v □  Tab[ i ] ;  // l'élément frontière : ai
    j □  i ;        // le rang de l'élément frontière
     Tantque Tab[ j-1 ]> v faire//on travaille sur la partie déjà triée (a1, a2, ... , ai)
           Tab[ j ]  □ Tab[ j-1 ]; // on décale l'élément
            j □  j-1;                // on passe au rang précédent
     FinTant ;
     Tab[ j ]  □  v //on recopie ai dans la place libérée
  fpour


Fin Tri_Insertion
```

On utilise une sentinelle placée dans la cellule de rang 0 du tableau, comme le type d'élément du tableau est un **int**, nous prenons comme valeur de la sentinelle une valeur négative très grande par rapport aux valeurs des éléments du tableau; par exemple le plus petit élément du type int, soit la valeur Integer.MIN_VALUE.

Solution en Java
Tri par insertion sur un tableau d'entiers

La méthode Java implantant l'algorithme de tri par insertion :

```
static void TriInsert ( ) {
// sous-programme de Tri par insertion :
   int n = table.length-1;
   for ( int i = 2; i <= n; i++) {
      int v = table[i];
      int j = i;
      while (table[ j-1 ] > v) {//on travaille sur la partie déjà triée (a1, a2, … , ai)
         table[ j ] = table[ j-1 ]; // on décale l'élément
         j = j-1;                   // on passe au rang précédent
      } ;
      table[ j ]  = v ; //on recopie ai dans la place libérée
   }
}
```

Une classe contenant cette méthode et la testant :

```
class ApplicationTriInsert  {
  // le tableau à trier : 20 éléments
 static int[] table = new int[20] ;

  static void TriInsert ( ) {
    /* sous-programme de Tri par insertion :

  static void AfficherTable ( ) {
  //-- Affichage du tableau
    int n = table.length-1;
    for ( int i = 0; i <= n; i++)
       System.out.print(table[i]+" , ");
    System.out.println();
  }

  static void InitTable  ( ) {
  //-- remplissage aléatoire du tableau
    int n = table.length-1;
    for ( int i = 1; i <= n ; i++)
       table[i] = (int)(Math.random()*100);
    table[0] = -Integer.MAX_VALUE; //sentinelle à l'indice 0
  }

  public static void main(String[ ] args) {
    InitTable ( );
    System.out.println("Tableau initial :");
    AfficherTable ( );
    TriInsert ( );
    System.out.println("Tableau une fois trié :");
    AfficherTable ( );
  }
}
```

Algorithme

Recherche linéaire dans une table non triée

Objectif : Ecrire un programme Java effectuant une recherche séquentielle dans un tableau linéaire (une dimension) non trié

TABLEAU NON TRIE

Spécifications de l'algorithme :

- Soit **t** un tableau d'entiers de **1..n** éléments non rangés.

- On recherche le rang (la place) de l'élément **Elt** dans ce tableau. L'algorithme renvoie le rang (la valeur -1 est renvoyée lorsque l'élément **Elt** n'est pas présent dans le tableau **t**)

Version **Tantque** avec "**et alors**" (opérateur et optimisé)

```
 i □  1 ;
Tantque (i □ n) et alors (t[i] □ Elt) faire
      i □  i+1
finTant;
si  i □ n  alors rang □  i
sinon rang □  -1
Fsi
```

Version **Tantque** avec "**et**" (opérateur et non optimisé)

```
 i □  1 ;
Tantque (i < n) et (t[i] □ Elt) faire
      i □  i+1
finTant;
si  t[i] = Elt alors rang □  i
sinon rang □  -1
Fsi
```

Version **Tantque** avec sentinelle en fin de tableau (rajout d'une cellule)

```
t[n+1] □ Elt ; // sentinelle rajoutée
i □ 1 ;
Tantque (i □ n) et alors (t[i] □ Elt) faire
      i □ i+1
finTant;
si i □ n alors rang □ i
sinon rang □ -1
Fsi
```

Version **Pour** avec instruction de sortie (**Sortirsi**)

```
pour i □ 1 jusquà n faire
  Sortirsi t[i] = Elt
fpour;
si i □ n alors rang □ i
sinon rang □ -1
Fsi
```

Traduire chacune des quatre versions sous forme d'une méthode Java.

Proposition de squelette de classe Java à implanter :

```
class ApplicationRechLin {

      static int max = 20;
      static int[ ] table = new int[max] ; //20 cellules à examiner de 1 à 19
      static int[ ] tableSent = new int[max+1] ; // le tableau à examiner de 1 à 20


 static void AfficherTable (int[ ] t ) {
         // Affichage du tableau
       int n = t.length-1;
       for ( int i = 1; i <= n; i++)
          System.out.print(t[i]+" , ");
       System.out.println();
 }


 static void InitTable ( ) {
         // remplissage aléatoire du tableau
       int n = table.length-1;
       for ( int i = 1; i <= n; i++) {
```

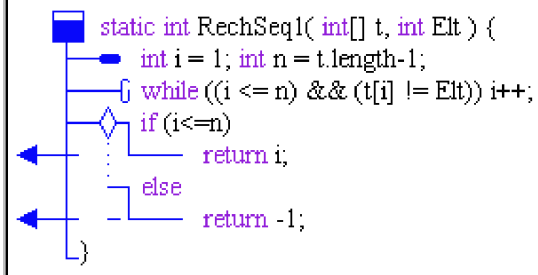
```
            table[i] = (int)(Math.random( )*100);
            tableSent[i] = table[i];
          }
    }

    static int RechSeq1( int[ ] t, int Elt ) {

    }

    static int RechSeq2( int[ ] t, int Elt ) {

    }

    static int RechSeq3( int[ ] t, int Elt ) {

    }

   static int RechSeq4( int[ ] t, int Elt ) {

    }

   public static void main(String[ ] args) {
      InitTable ( );
      System.out.println("Tableau initial :");
      AfficherTable (table );
      int x = Readln.unint(), rang;
      //appeler ici les méthodes de recherche…
      if (rang > 0)
        System.out.println("Elément "+x+" trouvé en : "+rang);
      else  System.out.println("Elément "+x+" non trouvé !");
    }
}
```

Solution en Java

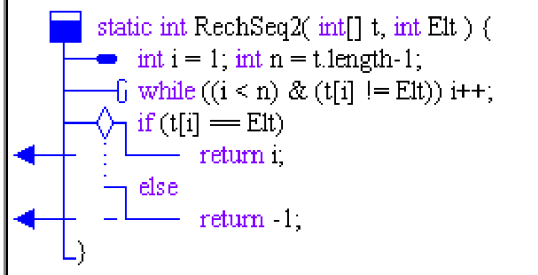
Recherche linéaire dans une table non triée

Les différentes méthodes Java implantant les 4 versions d'algorithme de recherche linéaire (table non triée) :

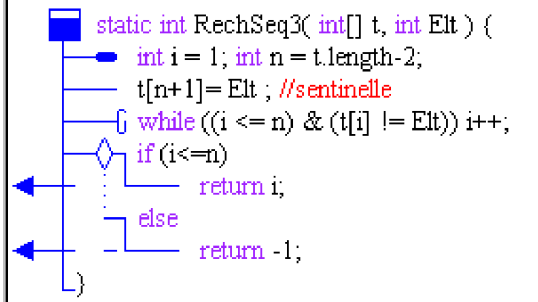
Version **Tantque** avec "**et alors**" (optimisé)



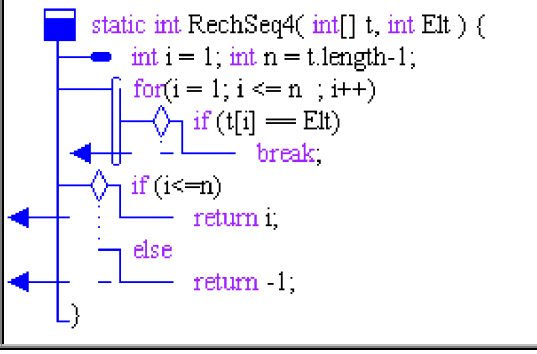
Version **Tantque** avec "**et**" (non optimisé)



Version **Tantque** avec sentinelle à la fin



Version **Pour** avec break




```
class ApplicationRechLin  {

      static int max = 20;
      static int[ ] table = new int[max] ; //20 cellules à examiner de 1 à 19
      static int[ ] tableSent = new int[max+1] ; // le tableau à examiner de 1 à 20


 static void AfficherTable (int[ ] t ) {
          // Affichage du tableau
        int n = t.length-1;
        for ( int i = 1; i <= n; i++)
          System.out.print(t[i]+" , ");
        System.out.println();
 }
```

```
  static void InitTable  ( ) {
          // remplissage aléatoire du tableau
        int n = table.length-1;
        for ( int i = 1; i <= n; i++) {
          table[i] = (int)(Math.random( )*100);
          tableSent[i] = table[i];
        }
  }

  static int RechSeq1( int[ ] t, int Elt ) { … }

  static int RechSeq2( int[ ] t, int Elt ) { … }

  static int RechSeq3( int[ ] t, int Elt ) { … }

 static int RechSeq4( int[ ] t, int Elt ) { … }

  public static void main(String[ ] args) {
     InitTable ( );
     System.out.println("Tableau initial :");
     AfficherTable (table );
     int x = Readln.unint(), rang;
     //rang = RechSeq1( table, x );
     //rang = RechSeq2( table, x );
     //rang = RechSeq3( tableSent, x );
     rang = RechSeq4( table, x );
     if (rang > 0)
        System.out.println("Elément "+x+" trouvé en : "+rang);
     else  System.out.println("Elément "+x+" non trouvé !");
   }
}
```

Algorithme

Recherche linéaire dans une table déjà triée

Objectif : Ecrire un programme Java effectuant une recherche séquentielle dans un tableau linéaire (une dimension) déjà trié.

TABLEAU DEJA TRIE

Spécifications de l'algorithme :

- Soit **t** un tableau d'entiers de **1..n** éléments rangés par ordre croissant par exemple.
- On recherche le rang (la place) de l'élément **Elt** dans ce tableau. L'algorithme renvoie le rang (la valeur -1 est renvoyée lorsque l'élément **Elt** n'est pas présent dans le tableau **t**)

On peut reprendre sans changement les algorithmes précédents travaillant sur un tableau non trié.

On peut aussi utiliser le fait que le dernier élément du tableau est **le plus grand élément** et s'en servir comme une sorte de **sentinelle**. Ci-dessous deux versions utilisant cette dernière remarque.

Version **Tantque** :

```
si t[n] < Elt alors rang □    -1
sinon
   i □   1 ;
   Tantque t[i] < Elt faire
      i □    i+1;
    finTant;
    si t[i] □ Elt alors rang □   i
    sinon rang □  -1 Fsi
Fsi
```

Version **pour** :

```
si t[n] < Elt alors rang □ -1
sinon
   pour i □ 1 jusquà n-1 faire
     Sortirsi t[i] □ Elt    // sortie de la boucle
   fpour;
   si t[i] □ Elt alors rang □ i
   sinon rang □ -1 Fsi
Fsi
```

Ecrire chacune des méthodes associées à ces algorithmes (prendre soin d'avoir trié le tableau auparavant par exemple par une méthode de tri), squelette de classe proposé :

```
class ApplicationRechLinTrie  {

   static int[ ] table = new int[20] ; //20 cellules à examiner de 1 à 19


   static void AfficherTable (int[ ] t ) {
      // Affichage du tableau
     int n = t.length-1;
     for ( int i = 1; i<=n; i++)
       System.out.print(t[i]+" , ");
     System.out.println( );
   }
  static void InitTable  ( ) {
      // remplissage aléatoire du tableau
     int n = table.length-1;
     for ( int i = 1; i<=n; i++) {
       table[i] = (int)(Math.random( )*100);
     }
   }
  static void TriInsert ( ) { // sous-programme de Tri par insertion ...  }


  static int RechSeqTri1( int[] t, int Elt ) {…}


  static int RechSeqTri2( int[] t, int Elt ) {…}

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

Solution en Java
Recherche linéaire dans une table déjà triée

Les deux méthodes Java implantant les 2 versions d'algorithme de recherche linéaire (table déjà triée) :

```
static int RechSeqTri1( int[] t, int Elt ) {
  int i = 1; int n = t.Length-1;
  if (t[n] >= Elt) {
    while (t[i] < Elt) i++;
    if (t[i] == Elt)
      return  i ;
  }
  return  -1;
}
```

```
static int RechSeqTri2( int[] t, int Elt )
{
  int i = 1; int n = t.Length-1;
  if (t[n] >= Elt) {
    for (i = 1; i <= n; i++)
      if (t[i] == Elt)
        return  i;
  }
  return  -1;
}
```

La méthode main de la classe ApplicationRechLinTrie :

public static void main(String[ ] args)

```
{
        InitTable ( );
        System.out.println("Tableau initial :");
        AfficherTable (table );
        TriInsert ( );
        System.out.println("Tableau trié :");
        AfficherTable (table );
        int x = Readln.unint( ), rang;
        //rang = RechSeqTri1( table, x );
        rang = RechSeqTri2( table, x );
        if (rang > 0)
           System.out.println("Elément "+x+" trouvé en : "+rang);
        else  System.out.println("Elément "+x+" non trouvé !");

}
```

Algorithme
Liste triée de noms en Java

Objectif : Effectuer un travail de familiarisation avec la structure de liste dynamique adressable triée correspondant à la notion de tableau dynamique trié. Ce genre de structure cumule les avantages d'une structure de liste linéaire (insertion, ajout,...) et d'un tableau autorisant les accès direct par un index.

La classe concernée se dénomme Vector, elle hérite de la classe abstraite AbstractList et implémente l'interface List (public class Vector **extends** AbstractList **implements** List **)**

Nous allons utiliser un Vector pour implanter une liste triée de noms, les éléments contenus dans la liste sont des chaînes de caractères (des noms)..

Question n°1:

Codez la méthode "initialiser" qui permet de construire la liste suivante :
Liste = (voiture, terrien, eau, pied, traineau, avion, source, terre, xylophone, mer, train, marteau).

Codez la méthode "ecrire" qui permet d'afficher le contenu de la liste et qui produit l'affichage suivant :

voiture, terrien, eau, pied, traineau, avion, source, terre, xylophone, mer, train,
marteau,
Taille de la liste chaînée = 12

squelette proposé pour chaque méthode :
static void initialiser (Vector L) {....}
static void ecrire(Vector L) {....}

Remarque importante :
Une entité de classe Vector est un objet. un paramètre Java de type objet est une référence, donc nous n'avons pas le problème du passage par valeur du contenu d'un objet. En pratique cela signifie que lorsque le paramètre est un objet, il est à la fois en entrée et en sortie. Ici le Vector L est modifié par toute action interne effectuée sur lui dans les méthodes "initialiser" et "ecrire".

Question n °2:

Ecrire une méthode permettant de trier la liste des noms par odre alphabétique croissant en utilisant l'algorithme de **tri par sélection**.

On donne l'algorithme de tri par selection suivant :

Algorithme Tri_Selection
local: m, i , j , **n**, temp □ Entiers naturels
Entrée : Tab □ Tableau d'Entiers naturels de **1** à **n** éléments
Sortie : Tab □ Tableau d'Entiers naturels de **1** à **n** éléments

début
pour i **de** 1 **jusquà** **n-1** **faire** *// recommence une sous-suite*
m □ i ; *// i est l'indice de l'élément frontière **a**i = Tab[i]*
pour j **de** i+1 **jusquà** **n** **faire** *// (**a**i+1, **a**2, ... , **a**n)*
si Tab[j] < Tab[m] **alors** *// **a**j est le nouveau minimum partiel*
m □ j ; *// indice mémorisé*
Fsi
fpour;
temp □ Tab[m] ;
Tab[m] □ Tab[i] ;
Tab[i] □ temp *//on échange les positions de **a**i et de **a**j*
fpour
Fin Tri_Selection

squelette proposé pour la méthode :
static void triSelect (Vector L) {....}

Question n°3:

Ecrire une méthode permettant d'insérer un nouveau nom dans une liste déjà triée, selon l'algorithme proposé ci-dessous :

L : Liste de noms déjà triée,
Elt : le nom à insérer dans la liste L.
taille(L) : le nombre d'éléments de L

début
si (la liste L est vide) **ousinon** (dernierElement de la liste L □ Elt) **alors**
ajouter Elt en fin de liste L
sinon
pour i □ 0 **jusquà** taille(L)-1 **faire**
si Elt □ Element de rang i de L **alors**
insérer Elt à cette position ;
sortir
fsi
fpour

fsi
fin

squelette proposé pour la méthode :

static void inserElem (Vector L, String Elt) {....}

Contenu proposé de la méthode main avec les différents appels :

```
static void main(String[] Args) {
    Vector Liste = new Vector();
  //---> contenu de la Liste - initialisation :
    initialiser(Liste);
    ecrire(Liste);

  //---> Tri de la liste
    System.out.println("\nListe une fois triée : ");
    triSelect(Liste);
    ecrire(Liste);

  //---> Insérer un élément dans la liste triée
    String StrInserer ="trainard";
    System.out.println("\nInsertion dans la liste de : "+StrInserer);
    inserElem(Liste, StrInserer);
    ecrire(Liste);

  //---> Contenu de la Liste - boolean remove(Object x) :
    System.out.println("\nListe.remove('pied') : ");
    Liste.remove("pied");
    ecrire(Liste);
}
```

Voici ci-dessous les méthodes de la classe Vector, principalement utiles à la manipulation d'une telle liste:

Classe **Vector** :

boolean add(Object elem)	Ajoute l'élément "elem" à la fin du Vector et augmente sa taille de un.
void add(int index, Objectelem)	Ajoute l'élément "elem" à la position spécifiée par index et augmente la taille du Vector de un.
void clear( )	Efface tous les éléments présents dans le Vector et met sa taille à zéro.
Object firstElement( )	Renvoie le premier élément du Vector (l'élément de rang 0). Il faudra le transtyper selon le type d'élément du Vector.
Object get(int index)	Renvoie l'élément de rang index du Vector. Il faudra le

	transtyper selon le type d'élément du Vector.
int indexOf(Object elem)	Cherche le rang de la première occurence de l'élément "elem" dans le Vector. Renvoie une valeur comprise entre **0** et **size()-1** si "elem" est trouvé, revoie **-1** sinon.
void insertElementAt(Object elem, intindex)	Insère l'élément "elem" à la position spécifiée par index et augmente la taille du Vector de un.
boolean isEmpty( )	Teste si le Vector n'a pas d'éléments (renvoie **true**); renvoie **false** s'il contient au moins un élément.
Object lastElement( )	Renvoi le dernier élément du Vector (l'élément de rang **size()-1**). Il faudra le transtyper selon le type d'élément du Vector.
Object remove(int index)	Efface l'élément de rang index du Vector. La taille du Vector diminue de un.
boolean remove(Object elem)	Efface la première occurence de l'élément elem du Vector, la taille du Vector diminue alors de un et renvoie true. Si elem n'est pas trouvé la méthode renvoie false et le Vector n'est pas touché.
int size( )	Renvoie la taille du Vector (le nombres d'éléments contenus dans le Vector).

Solution en Java
Liste triée de noms en Java

Question n°1:

Le sous programme Java implantant la méthode "initialiser" construisant la liste :

```
static void initialiser(Vector L) {
    L.add("voiture" );
    L.add("terrien" );
    L.add("eau" );
    L.add("pied" );
    L.add("traineau" );
    L.add("avion" );
    L.add("source" );
    L.add("terre" );
    L.add("xylophone" );
    L.add("mer" );
    L.add("train" );
    L.add("marteau" );
}
```

Le sous programme Java implantant la méthode "ecrire" affichant le contenu de la liste :

```
static void ecrire(Vector L) {
    for(int i=0; i<L.size(); i++) {
        System.out.print(L.get(i)+", ");
    }
    System.out.println("\nTaille de la liste chaînée = "+L.size());
}
```

Question n°2:

La méthode "triSelect" utilisant l'algorithme de tri par sélection sur un tableau d'entiers :

```
static void TriSelect ( ) {
// sous-programme de Tri par sélection :
    int n = table.length-1;
    for ( int i = 1; i <= n-1; i++)
    { // recommence une sous-suite
        int m = i; // i est l'indice de l'élément frontière ai = table[ i ]
        for ( int j = i+1; j <= n; j++)   // (ai+1, a2, ... , an)
            if (table[ j ] < table[ m ]) // aj est le nouveau minimum partiel
                m = j ; // indice mémorisé
      //on échange les positions de ai et de aj :
        int temp = table[ m ];
        table[ m ] = table[ i ];
        table[ i ]= temp;
    }
}
```

Nous allons construire à partir de ce modèle la méthode **static void** triSelect (Vector L)

{....}qui travaille sur un Vector, en utilisant les remarques suivantes.

Remarques :

- Nous devons ranger des noms par ordre alphabétique croissant et non des entiers, les noms sont des String, nous devons considérer le Vector comme un tableau de String pour pouvoir le trier.
- **table[i]** de l'algorithme (accès au ième élément) est implanté en Vector par **L.get(i).**
- Comme la méthode "Object get(**int** index)" renvoie un Object nous transtypons **L.get(i)** en type String qui est un descendant d'Object, grâce à la méthode de classe valueOf de la classe String "**static** String valueOf(Object obj)", ce qui s'écrit ici : String.valueOf(**L.get(i)**).
- Les opérateurs de comparaisons <, > etc… ne prennent pas en charge le type String comme en Delphi, il est donc nécessaire de chercher dans la liste des méthodes de la classe String une méthode permettant de comparer lexicographiquement deux String.
- Pour comparer deux String (**s1** < **s2**)on trouve la méthode **compareTo** de la classe String :

  String.valueOf(L.get(j)).**compareTo**(String.valueOf(L.get(m))) < 0.

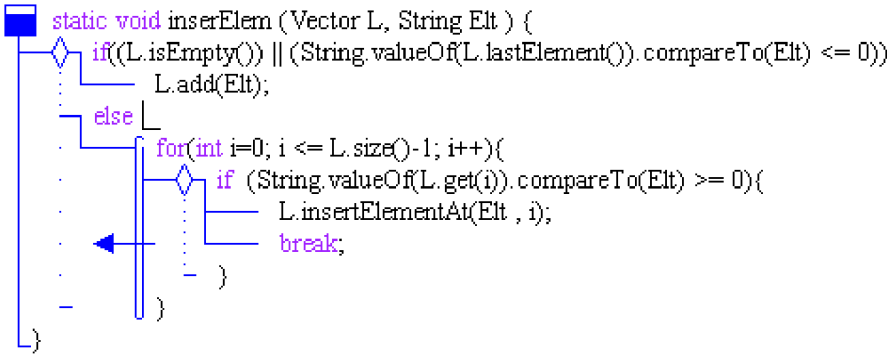
  Implantera la comparaison : **table[j]** < **table[m]**

Ce qui nous donne la méthode triSelect (Vector L) suivante :

```
static void triSelect (Vector L ) {
// sous-programme de Tri par sélection de la liste
   int n = L.size()-1;
   for ( int i = 0; i <= n-1; i++)
   { // recommence une sous-suite
      int m = i; // i est l'indice de l'élément frontière ai = table[ i ]
      for ( int j = i+1; j <= n; j++)   // (ai+1, a2, … , an)
         if ( String.valueOf(L.get(j)).compareTo(String.valueOf(L.get(m))) < 0 )
            m = j ; // indice mémorisé
      String temp = String.valueOf(L.get(m));  // int temp = table[ m ];
      L.set(m, L.get(i));  //table[ m ] = table[ i ];
      L.set(i, temp);  //table[ i ]= temp;
   }
}
```

Question n°3:

La méthode "inserElem" utilisant l'algorithme d'insertion d'un élément dans une liste triée :



Explications :

Voici les traductions utilisées pour implanter l'algorithme d'insertion :

Algorithme	Java Vector et String
la liste L est vide	L.isEmpty( )
ousinon (ou optimisé)	||
dernierElement de la liste L	L.lastElement( ) . Transtypé en String par : String.valueOf(L.lastElement( ))
dernierElement de la liste L □ Elt	String.valueOf(L.lastElement( )).compareTo(Elt) <= 0
ajouter Elt en fin de liste L	L.add(Elt)
taille(L)	L.size( )
Element de rang i de L	String.valueOf(L.get(i))
Elt □ Element de rang i de L	String.valueOf(L.get(i)).compareTo(Elt) >= 0
insérer Elt à cette position	L.insertElementAt(Elt , i)
sortir	break

Une classe complète permettant les exécutions demandées :

```
import java.util.Vector; // nécessaire à l'utilisation des Vector

 class ApplicationListeSimple

{
```

```
    static void triSelect (Vector L ) {
      // sous-programme de Tri par sélection de la liste
      int n = L.size()-1;
      for ( int i = 0; i <= n-1; i++)
      { // recommence une sous-suite
        int m = i; // i est l'indice de l'élément frontière ai = table[ i ]
        for ( int j = i+1; j <= n; j++)   // (ai+1, a2, ... , an)
          if ( String.valueOf(L.get(j)).compareTo(String.valueOf(L.get(m))) < 0 )
            m = j ; // indice mémorisé
        String temp = String.valueOf(L.get(m));  // int temp = table[ m ];
        L.set(m, L.get(i));  //table[ m ] = table[ i ];
        L.set(i, temp);  //table[ i ]= temp;
      }
    }


 static void inserElem (Vector L, String Elt ) {
  if((L.isEmpty()) || (String.valueOf(L.lastElement()).compareTo(Elt) <= 0))
    L.add(Elt);
  else
    for(int i=0; i <= L.size()-1; i++){
      if  (String.valueOf(L.get(i)).compareTo(Elt) >= 0){
        L.insertElementAt(Elt , i);
        break;
      }
    }
}


 static void ecrire(Vector L) {
  for(int i=0; i<L.size(); i++) {
    System.out.print(L.get(i)+", ");
  }
  System.out.println("\nTaille de la liste chaînée = "+L.size());
}


 static void initialiser(Vector L) {
  L.add("voiture" );
  L.add("terrien" );
  L.add("eau" );
  L.add("pied" );
  L.add("traineau" );
  L.add("avion" );
  L.add("source" );
  L.add("terre" );
  L.add("xylophone" );
  L.add("mer" );
  L.add("train" );
  L.add("marteau" );
}
```

```
    public static void main(String[] Args) {
      Vector Liste = new Vector( ); //création obligatoire d'un objet de classe Vector
     //---> contenu de la Liste - initialisation :
      initialiser(Liste);
      ecrire(Liste);

          //---> Tri de la liste :
           System.out.println("\nListe une fois triée : ");
           triSelect(Liste);
           ecrire(Liste);

          //---> Insérer un élément dans la liste triée :
           String StrInserer ="trainard";
           System.out.println("\nInsertion dans la liste de : "+StrInserer);
           inserElem(Liste, StrInserer);
           ecrire(Liste);

          //---> Contenu de la Liste - boolean remove(Object x) :
           System.out.println("\nListe.remove('pied') : ");
           Liste.remove("pied");
           ecrire(Liste);
        }
}
```

Exécution de cette classe :

voiture, terrien, eau, pied, traineau, avion, source, terre, xylophone, mer, train, marteau,
Taille de la liste chaînée = 12

Liste une fois triée :
avion, eau, marteau, mer, pied, source, terre, terrien, train, traineau, voiture, xylophone,
Taille de la liste chaînée = 12

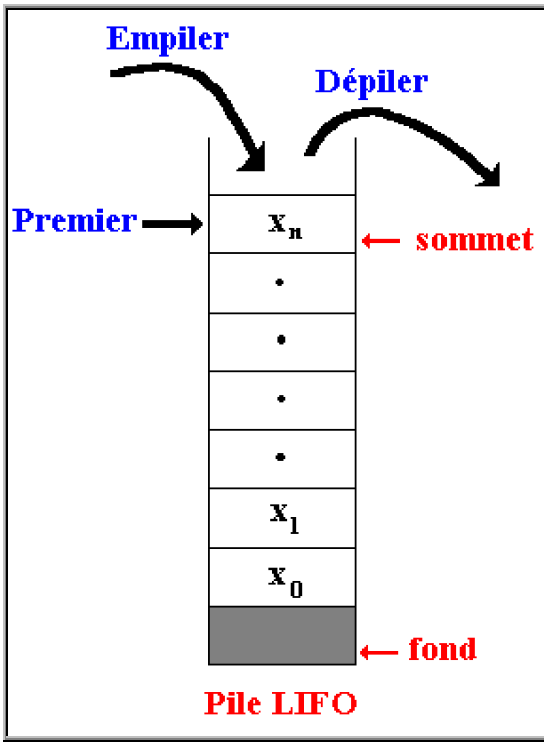
Insertion dans la liste de : trainard
avion, eau, marteau, mer, pied, source, terre, terrien, train, trainard, traineau, voiture, xylophone,
Taille de la liste chaînée = 13

Liste.remove('pied') :
avion, eau, marteau, mer, source, terre, terrien, train, trainard, traineau, voiture, xylophone,
Taille de la liste chaînée = 12

Algorithme
Structure de donnée de pile LIFO

Objectif : Nous implantons en Java une structure de pile LIFO (Last In First Out) fondée sur l'utilisation d'un objet de classse LinkedList. Nous construisons une pile LIFO de chaînes de caractères.

Rappel des spécifications d'une pile LIFO :



Opérateurs de base sur une pile LIFO :

sommet pointe vers l'élément en haut de pile, fond sert de sentinelle à la pile si nécessaire.

Dépiler ([X0, X1,..., Xn, Xn+1]) -->
Pile = [X0, X1,..., Xn] , Xn+1

Empiler([X0, X1,..., Xn] , Xn+1) -->
Pile = [X0, X1,..., Xn, Xn+1]

Premier([X0, X1,..., Xn]) = Xn

EstVide([X0, X1,..., Xn]) = **false**
EstVide([ ]) = **true** *(sommet = fond)*

Notre pile LIFO doit contenir des noms (chaînes de caractères donc utilisation des String).

La classe LinkedList est une structure dynamique (non synchronizée) qui ressemble à la classe Vector, mais qui est bien adaptée à implanter les piles et les files car elle contient des références de type Object et les String héritent des Object.

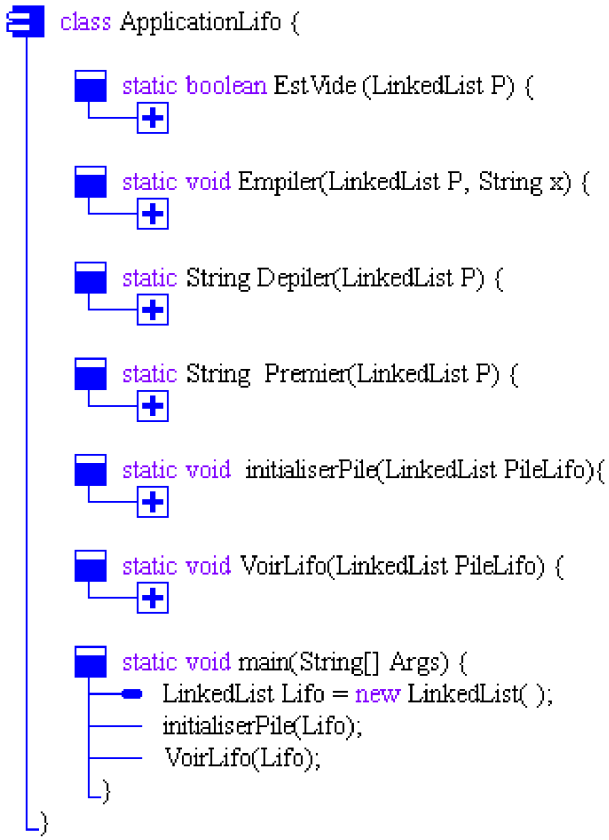
Proposition de squelette de classe Java algorithmique :

Nous utilisons un objet de classe **LinkedList** pour représenter une **pile LIFO**, elle sera passée comme paramètre dans les méthodes qui travaillent sur cet objet :

static boolean EstVide (LinkedList P)	tester si la pile P est vide

static void Empiler(LinkedList P, String x)	Empiler dans la pile P le nom x.
static String Depiler(LinkedList P)	Dépiler la pile P.
static String Premier(LinkedList P)	Renvoyer l'élément au sommet de la pile P.
static void initialiserPile(LinkedList P)	Remplir la pile P avec des noms.
static void VoirLifo(LinkedList P)	Afficher séquentiellement le contenu de P.

Complétez la classe ci-dessous et ses méthodes :



Voici ci-dessous les méthodes principalement utiles à la manipulation d'une telle liste:

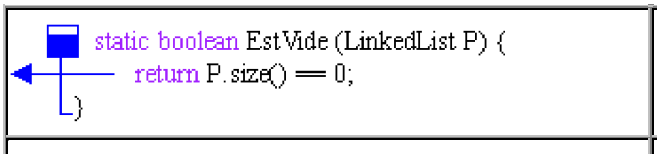
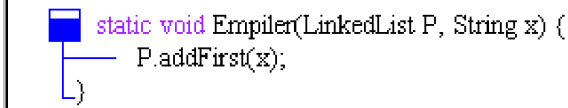
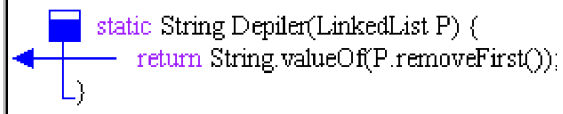
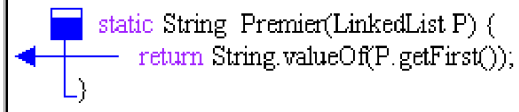
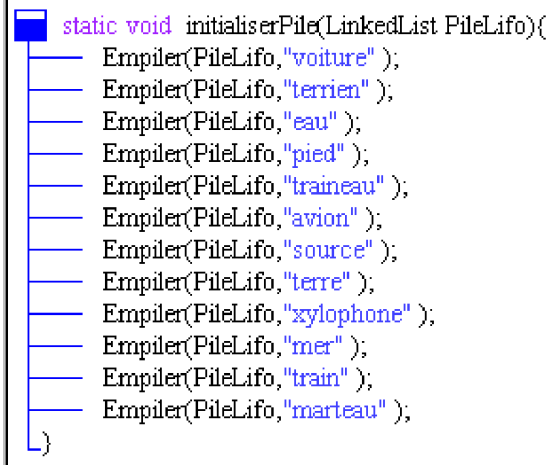
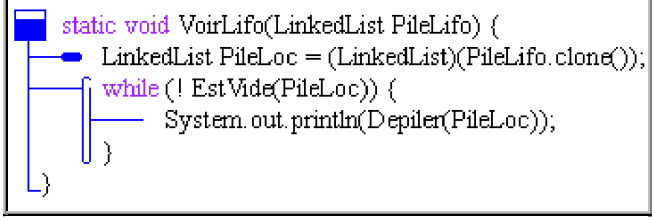
Classe **LinkedList** :

boolean add(Object elem)	Ajoute l'élément "elem" à la fin de la LinkedList et augmente sa taille de un.
void add(int index, Object elem)	Ajoute l'élément "elem" à la position spécifiée par index et augmente la taille de la LinkedList de un.
void clear( )	Efface tous les éléments présents dans la LinkedList et met sa taille à zéro.

Object getFirst()	Renvoie le premier élément de la LinkedList (l'élément en tête de liste, rang=0). Il faudra le transtyper selon le type d'élément de la LinkedList.
Object getLast()	Renvoie le dernier élément de la LinkedList (l'élément en fin de liste, rang=**size()-1**). Il faudra le transtyper selon le type d'élément de la LinkedList.
Object get(int index)	Renvoie l'élément de rang index de la LinkedList. Il faudra le transtyper selon le type d'élément de la LinkedList.
int indexOf(Object elem)	Cherche le rang de la première occurence de l'élément "elem" dans le Vector. Renvoie une valeur comprise entre **0** et **size()-1** si "elem" est trouvé, revoie **-1** sinon.
void addFirst(Object elem)	Insère l'élément "elem" en tête de la LinkedList (rang=0).
void addLast(Object elem)	Ajoute l'élément "elem" à la fin de la LinkedList et augmente sa taille de un.
boolean isEmpty( )	Teste si la LinkedList n'a pas d'éléments (renvoie **true**); renvoie **false** si elle contient au moins un élément.
Object remove(int index)	Efface l'élément de rang index de la LinkedList. La taille de la LinkedList diminue de un.
boolean remove(Object elem)	Efface la première occurence de l'élément elem de la LinkedList, la taille de la LinkedList diminue alors de un et renvoie true. Si elem n'est pas trouvé la méthode renvoie false et la LinkedList n'est pas touché.
Object removeFirst()	Efface et renvoie le premier élément (rang=0) de la LinkedList. Il faudra le transtyper selon le type d'élément de la LinkedList.
Object removeLast()	Efface et renvoie le dernier (rang=**size()-1**) élément de la LinkedList. Il faudra le transtyper selon le type d'élément de la LinkedList.
int size( )	Renvoie la taille de la LinkedList (le nombres d'éléments contenus dans la LinkedList).

Solution en Java
Structure de donnée de pile LIFO

Les méthodes s'appliquant à la pile LIFO :

	tester si la pile P est vide
	Empiler dans la pile P le nom x.
	Dépiler la pile P.
	Renvoyer l'élément au sommet de la pile P.
	Remplir la pile PileLifo avec des noms.
	Afficher séquentiellement le contenu de PileLifo.

Une classe complète permettant l'exécution des méthodes précédentes :

```
import java.util.LinkedList;

      class ApplicationLifo {

        static boolean EstVide (LinkedList P) {
          return P.size() == 0;
        }


        static void Empiler(LinkedList P, String x) {
          P.addFirst(x);
        }


        static String Depiler(LinkedList P) {
          return String.valueOf(P.removeFirst());
        }


        static String  Premier(LinkedList P) {
          return String.valueOf(P.getFirst());
        }


        static void  initialiserPile(LinkedList PileLifo){
          Empiler(PileLifo,"voiture" );
          Empiler(PileLifo,"terrien" );
          Empiler(PileLifo,"eau" );
          Empiler(PileLifo,"pied" );
          Empiler(PileLifo,"traineau" );
          Empiler(PileLifo,"avion" );
          Empiler(PileLifo,"source" );
          Empiler(PileLifo,"terre" );
          Empiler(PileLifo,"xylophone" );
          Empiler(PileLifo,"mer" );
          Empiler(PileLifo,"train" );
          Empiler(PileLifo,"marteau" );
        }


        static void VoirLifo(LinkedList PileLifo) {
          LinkedList PileLoc = (LinkedList)(PileLifo.clone());
          while (! EstVide(PileLoc)) {
            System.out.println(Depiler(PileLoc));
          }
        }
```

```
    public static void main(String[ ] Args) {
      LinkedList Lifo = new LinkedList( );
      initialiserPile(Lifo);
      VoirLifo(Lifo);
    }
}
```

Exercice

lire et écrire un enregistrement dans un fichier texte

Objectif : Nous implantons en Java une classe d'écriture dans un fichier texte d'informations sur un client et de lecture du fichier pour rétablir les informations initiales.

Chaque client est identifié à l'aide de quatre informations :

Numéro de client

Nom du client

Prénom du client

Adresse du client

Nous rangeons ces quatre informations dans le même enregistrement-client. L'enregistrement est implanté sous forme d'une ligne de texte contenant les informations relatives à un client, chaque information est séparée de la suivante par le caractère de séparation # .

Par exemple, les informations client suivantes :

Numéro de client = `12598`

Nom du client = `Dupont`

Prénom du client = `Pierre`

Adresse du client = `2, rue des moulins 37897 Thiers`

se touvent rangées dans un enregistrement constitué de quatre zones, sous la forme de la ligne de texte suivante :

```
12598#Dupont#Pierre#2, rue des moulins 37897 Thiers
```

Le fichier client se nommera "ficheclient.txt", vous écrirez les méthodes suivantes :

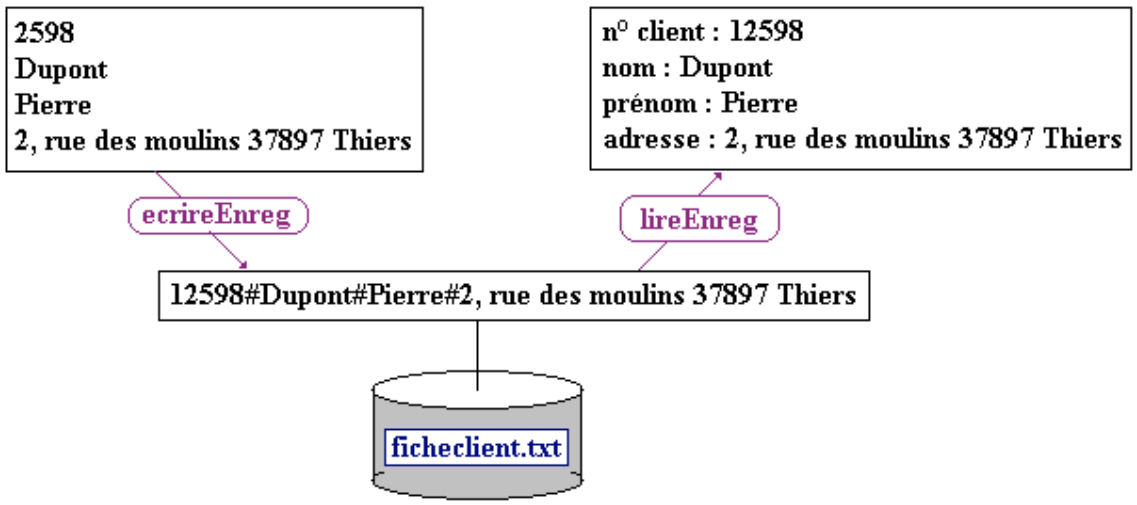
Signature de la méthode	Fonctionnement de la méthode
public static void ecrireEnreg (**String** nomFichier)	Ecrit dans le fichier client dont le nom est passé en paramètre, les informations d'un seul client sous forme d'un enregistrement (cf.ci-haut).
public static void lireEnreg (**String** nomFichier)	Lit dans le fichier client client dont le nom est passé en paramètre, un enregistrement et affiche sur la console les informations du client.
public static String[] extraitIdentite (**String**	Renvoie dans un tableau de String les 4 informations (n°, nom, prénom, adresse)

ligne)	contenues dans l'enregistrement passé en paramètre. Appelée par la méthode lireEnreg.
public static void Afficheinfo (**String**[ ] infos)	Affiche sur la console les informations du client contenues dans le tableau de String passé en paramètre. Appelée par la méthode lireEnreg.

Squelette java proposé pour la classe :

```
import java.io.*;

class AppliFichierTexte {
  private static final String separ="#";
  private static final String[] libelle=
          {"n° client : ","nom : ","prénom : ","adresse : "};

  public static void ecrireEnreg(String nomFichier) {

  public static String[] extraitIdentite(String ligne){

  public static void Afficheinfo(String[] infos){

  public static void lireEnreg(String nomFichier) {

  public static void main(String[] x){
    ecrireEnreg("ficheclient.txt");//local
    lireEnreg("ficheclient.txt");
  }
}
```

Modèle des actions effectuées par les méthodes de la classe :



Exercice

Copier un fichier texte dans un autre fichier texte

Objectif : Nous implantons en Java une classe de recopie de tout le contenu d'un fichier texte dans un nouveau fichier texte clone du premier.

Le fichier source se nommera "fiche.txt", le fichier de destination clone se dénommera "copyfiche.txt", vous écrirez les 2 méthodes suivantes :

Signature de la méthode	Fonctionnement de la méthode
public static void copyFichier (**String** FichierSource, **String** FichierDest)	Copie le contenu du FichierSource dans le FichierDest.
public static void lireFichier (**String** nomFichier)	Lit tout le contenu d'un fichier client dont le nom est passé en paramètre, et affiche sur la console les informations de tout le fichier.

Squelette java proposé pour la classe :

```
import java.io.*;

class AppliCopyFichierTexte {

    public static void copyFichier(String FichierSource, String FichierDest) {

    public static void lireFichier(String nomFichier) {

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

Méthode main de la classe et actions :

```
public static void main(String[] x){
    System.out.println("Fichier initial : ");
    lireFichier("fiche.txt");
    copyFichier("fiche.txt","copyfiche.txt");
    System.out.println("Fichier copié : ");
    lireFichier("copyfiche.txt");
}
```

solution Java

lire et écrire un enregistrement dans un fichier texte

La classe **AppliFichierTexte** et ses membres :

```
import java.io.*;

class AppliFichierTexte {
  private static final String separ="#";
  private static final String[] libelle=
          ("n° client : ","nom : ","prénom : ","adresse : ");
    ******
   public static void main(String[] x){
      ecrireEnreg("ficheclient.txt");//local
      lireEnreg("ficheclient.txt");
    }
}

  public static void ecrireEnreg(String nomFichier) {
     try {
        FileWriter fluxwrite = new FileWriter(nomFichier);
        BufferedWriter out = new BufferedWriter(fluxwrite);
        out.write(12598+separ);
        out.write("Dupont"+separ);
        out.write("Pierre"+separ);
        out.write("2, rue des moulins 37897 Thiers");
        out.newLine( ); //écrit le eoln
        out.close( ); //sinon le fichier créé sur le disque est vide !!
     }
       catch (IOException err) {
          System.out.println( "Erreur : " + err );
       }
  }

  public static String[] extraitIdentite(String ligne){
     return ligne.split(separ);
  }

public static void Afficheinfo(String[] infos){
   for(int i=0; i<infos.length; i++)
      System.out.println(libelle[i]+infos[i]);
}

public static void lireEnreg(String nomFichier) {
   try {
      FileReader fluxread = new FileReader(nomFichier);
      BufferedReader in = new BufferedReader(fluxread);
      String Ligne;
      while( (Ligne = in.readLine()) != null) { //not eof type String (-1 autres cas)
         System.out.println("Enregistrement: "+Ligne);
         String[] info = extraitIdentite(Ligne);
         Afficheinfo(info);
      }
      in.close( );
   }
     catch (IOException err) {
        System.out.println( "Erreur : " + err );
     }
}
```

solution Java

Copier un fichier texte dans un autre fichier texte

La classe **AppliCopyFichierTexte** et ses membres :

```
import java.io.*;

class AppliCopyFichierTexte {

   public static void copyFichier(String FichierSource, String FichierDest) {

   public static void lireFichier(String nomFichier) {

   public static void main(String[] x){
      System.out.println("Fichier initial : ");
      lireFichier("fiche.txt");
      copyFichier("fiche.txt","copyfiche.txt");
      System.out.println("Fichier copié : ");
      lireFichier("copyfiche.txt");
   }
}
```

```
public static void copyFichier(String FichierSource, String FichierDest) {
   String Ligne;
   try {
      FileWriter fluxwrite = new FileWriter(FichierDest);
      BufferedWriter out = new BufferedWriter(fluxwrite);
      FileReader fluxread = new FileReader(FichierSource);
      BufferedReader in = new BufferedReader(fluxread);
      while( (Ligne = in.readLine()) != null) {
         out.write(Ligne);
         out.newLine();
      }
      out.close( ); //sinon le fichier créé sur le disque est vide !!
      in.close( );
   }
      catch (IOException err) {
         System.out.println( "Erreur : " + err );
      }
}
```

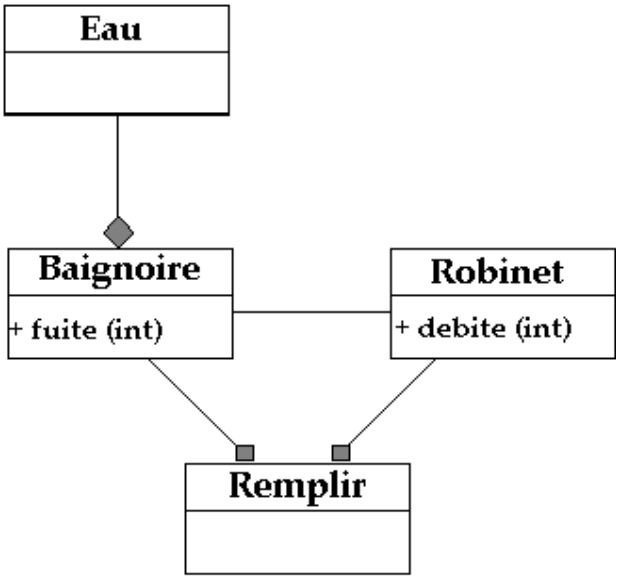
```
public static void lireFichier(String nomFichier) {
   try {
      FileReader fluxread = new FileReader(nomFichier);
      BufferedReader in = new BufferedReader(fluxread);
      String Ligne;
      //nul=eof dans le type String (-1 autres cas)
      while( (Ligne = in.readLine()) != null) {
         System.out.println(Ligne);
      }
   }
      catch (IOException err) {
         System.out.println( "Erreur : " + err );
      }
}
```

Exercice entièrement traité sur les threads

Enoncé et classes

Nous vous proposons de programmer une simulation du problème qui a tant mis à contribution les cerveaux des petits écoliers d'antan : *le problème du robinet qui remplit d'eau une baignoire qui fuit*. Nous allons écrire un programme qui simulera le remplissage de la baignoire par un robinet dont le débit est connu et paramétrable. La baignoire a une fuite dont le débit lui aussi est connu et paramétrable. Dès que la baignoire est entièrement vide l'on colmate la fuite et tout rentre dans l'ordre. On arrête le programme dès que la baignoire est pleine que la fuite soit colmatée ou non.

Nous choisissons le modèle objet pour représenter notre problème :



- Une classe **Eau** qui contient un champ static indiquant le volume d'eau actuel de l'objet auquel il appartient.
- Une classe **Baignoire** possédant un contenu (en litres d'eau) et une fuite qui diminue le volume d'eau du contenu de la baignoire.
- Une classe **Robinet** qui débite (augmente) le volume d'eau du contenu de la baignoire d'une quantité fixée.

Une première solution sans les threads

```
class Eau{
  static int volume;

  public Eau(int val){
    volume = val;
  }

}

class Robinet{
  public void debite(int quantite){
  }

}

class Baignoire {
  public static final int maximum=1000;
  public static Eau contenu = new Eau(0);

  public void fuite(int quantite ){
  }

}
```

Une classe **Remplir** qui permet le démarrage des actions : mise en place de la baignoire, du robinet, ouverture du robinet et fuite de la baignoire :

```
class Remplir {
  public static void main(String[] x){
    Baignoire UneBaignoire = new Baignoire();
    Robinet UnRobinet = new Robinet();
    UnRobinet.debite(50);
    UneBaignoire.fuite(20);
  }
}
```

Nous programmons les méthodes debite(**int** quantite) et fuite(**int** quantite) de telle sorte qu'elles afichent chacune l'état du contenu de la baignoire après que l'apport ou la diminution d'eau a eu lieu. Nous simulerons et afficherons pour chacune des deux méthodes 100 actions de base (100 diminutions pour la méthode fuite et 100 augmentations pour la méthode debite).

```
class Eau{
  static int volume;

  public Eau(int val){
    volume = val;
  }

}

class Robinet{
  public void debite(int quantite){
    for(int i=1; i<100; i++)
      if(Baignoire.contenu.volume < Baignoire.maximum){
        Baignoire.contenu.volume +=quantite;
        System.out.println("Contenu de la baignoire = "
                           +Baignoire.contenu.volume);
      }
      else {
        System.out.println("Baignoire enfin pleine !");
        break;
      }
  }

}
```

```
class Baignoire {
  public static final int maximum=1000;
  public static Eau contenu = new Eau(0);

  public void fuite(int quantite ){
    for(int i=1; i<100; i++)
      if(Baignoire.contenu.volume <= 0){
        System.out.println("Baignoire vide, on colmate la fuite !");
        break;
      }
      else if(Baignoire.contenu.volume >= Baignoire.maximum)
        break;
      else {
        Baignoire.contenu.volume -=quantite;
        System.out.println("Contenu de la baignoire = "
                           +Baignoire.contenu.volume);
      }
  }

}

class Remplir {
  public static void main(String[] x){
    Baignoire UneBaignoire = new Baignoire();
    Robinet UnRobinet = new Robinet();
    UnRobinet.debite(50);
    UneBaignoire.fuite(20);
  }
}
```

Résultats d'exécution :

Contenu de la baignoire = 50
Contenu de la baignoire = 100
Contenu de la baignoire = 150
Contenu de la baignoire = 200
Contenu de la baignoire = 250
Contenu de la baignoire = 300
Contenu de la baignoire = 350
Contenu de la baignoire = 400
Contenu de la baignoire = 450
Contenu de la baignoire = 500
Contenu de la baignoire = 550
Contenu de la baignoire = 600
Contenu de la baignoire = 650
Contenu de la baignoire = 700
Contenu de la baignoire = 750
Contenu de la baignoire = 800
Contenu de la baignoire = 850
Contenu de la baignoire = 900
Contenu de la baignoire = 950
Contenu de la baignoire = 1000
Baignoire enfin pleine !

---- operation complete.

Que s'est-il passé ?

La programmation séquentielle du problème n'a pas permis d'exécuter l'action de fuite de la baignoire puisque nous avons arrêté le processus dès que la baignoire était pleine. En outre nous n'avons pas pu simuler le remplissage et le vidage "simultanés" de la baignoire.

Nous allons utiliser deux threads (secondaires) pour rendre la simulation plus réaliste, en essayant de faire les actions de débit-augmentation et fuite-diminution en parallèle.

Deuxième solution avec des threads

Les objets qui produisent les variations du volume d'eau sont le robinet et la baignoire, ce sont eux qui doivent être "multi-threadés".

Reprenons pour cela la classe Robinet en la dérivant de la classe Thread, et en redéfinissant la méthode run( ) qui doit contenir le code de débit à exécuter en "parallèle" (le corps de la méthode **debite** n'a pas changé) :

```
class Robinet extends Thread {
    public void debite(int quantite){
        +
    public void run() {
        debite(50);
    }
}
```

De même en dérivant la classe Baignoire de la classe Thread, et en redéfinissant la méthode run( ) avec le code de fuite à exécuter en "parallèle" (le corps de la méthode **fuite** n'a pas changé) :

```
class Baignoire extends Thread {
    public static final int maximum=1000;
    public static Eau contenu = new Eau(0);
    public void fuite(int quantite ){
        +
    public void run() {
        fuite(20);
    }
}
```

Enfin la classe RemplirThread qui permet le démarrage des actions : mise en place de la baignoire, du robinet, puis lancement en parallèle de l'ouverture du robinet et de la fuite de la baignoire :

```
class RemplirThread {
    public static void main(String[] x){
        Baignoire UneBaignoire = new Baignoire();
        Robinet UnRobinet = new Robinet();
        UnRobinet.start();
        UneBaignoire.start();
    }
}
```

Résultats d'exécution obtenus :

Remplissage, contenu de la baignoire = 50 Remplissage, contenu de la baignoire = 100 Remplissage, contenu de la baignoire = 150 Remplissage, contenu de la baignoire = 200 Remplissage, contenu de la baignoire = 250	Remplissage, contenu de la baignoire = 900 Fuite, contenu de la baignoire = 880 Remplissage, contenu de la baignoire = 930 Fuite, contenu de la baignoire = 910 Fuite, contenu de la baignoire = 890

Remplissage, contenu de la baignoire = 300 Remplissage, contenu de la baignoire = 350 Remplissage, contenu de la baignoire = 400 Remplissage, contenu de la baignoire = 450 Remplissage, contenu de la baignoire = 500 Remplissage, contenu de la baignoire = 550 Remplissage, contenu de la baignoire = 600 Fuite, contenu de la baignoire = 580 Remplissage, contenu de la baignoire = 630 Fuite, contenu de la baignoire = 610 Remplissage, contenu de la baignoire = 660 Fuite, contenu de la baignoire = 640 Remplissage, contenu de la baignoire = 690 Fuite, contenu de la baignoire = 670 Remplissage, contenu de la baignoire = 720 Remplissage, contenu de la baignoire = 770 Remplissage, contenu de la baignoire = 820 Remplissage, contenu de la baignoire = 870 Fuite, contenu de la baignoire = 850	Fuite, contenu de la baignoire = 870 Fuite, contenu de la baignoire = 850 Fuite, contenu de la baignoire = 830 Fuite, contenu de la baignoire = 810 Fuite, contenu de la baignoire = 790 Fuite, contenu de la baignoire = 770 Fuite, contenu de la baignoire = 750 Fuite, contenu de la baignoire = 730 Fuite, contenu de la baignoire = 710 Fuite, contenu de la baignoire = 690 Remplissage, contenu de la baignoire = 740 Fuite, contenu de la baignoire = 720 Remplissage, contenu de la baignoire = 770 Remplissage, contenu de la baignoire = 820 Remplissage, contenu de la baignoire = 870 Remplissage, contenu de la baignoire = 920 Remplissage, contenu de la baignoire = 970 Remplissage, contenu de la baignoire = 1020 **Baignoire enfin pleine !**

---- operation complete.

Nous voyons que les deux threads s'exécutent cycliquement (mais pas d'une manière égale) selon un ordre non déterministe sur lequel nous n'avons pas de prise mais qui dépend de la java machine et du système d'exploitation, ce qui donnera des résultats différents à chaque nouvelle exécution. Le paragraphe suivant montre un exemple ou nous pouvons contraindre des threads de "dialoguer" pour laisser la place l'un à l'autre

variation sur les threads

Lorsque la solution adoptée est l'héritage à partir de la classe Thread, vous pouvez agir sur l'ordonnancement d'exécution des threads présents. Dans notre exemple utilisons deux méthodes de cette classe Thread :

- **void** setPriority (**int** newPriority)
- **static void** yield ( )

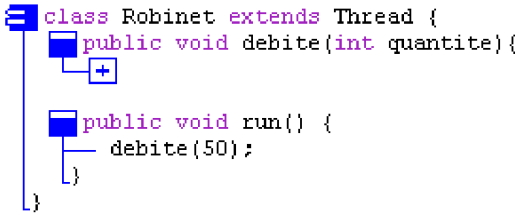
Privilégions le thread Robinet grâce à la méthode setPriority :

La classe Thread possède 3 champs **static** permettant d'attribuer 3 valeurs de priorités différentes, de la plus haute à la plus basse, à un thread indépendamment de l'échelle réelle du système d'exploitation sur lequelle travaille la Java Machine :

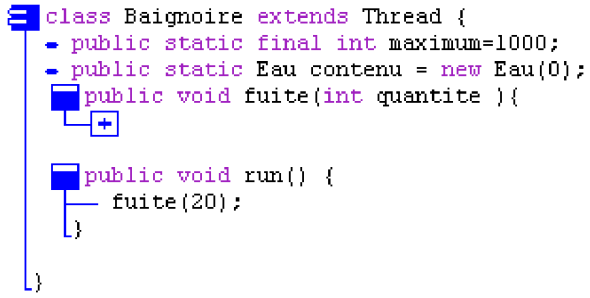
static int MAX_PRIORITY	La priorité **maximum** que peut avoir un thread.
static int MIN_PRIORITY	La priorité **minimum** que peut avoir un thread.
static int NORM_PRIORITY	La priorité **par défaut** attribuée à un thread.

La méthode setPriority appliquée à une instance de thread change sa priorité d'exécution.
Nous mettons l'instance UnRobinet à la priorité
maximum setPriority(Thread.MAX_PRIORITY) :

Classe Robinet sans changement



Classe Baignoire sans changement



Voici le changement de code dans la classe principale :

```
class RemplirThread {
     public static void main(String[] x){
       Baignoire UneBaignoire = new Baignoire();
       Robinet UnRobinet = new Robinet( );
       UnRobinet.setPriority ( Thread.MAX_PRIORITY );
       UnRobinet.start( );
       UneBaignoire.start( );
    }
}
```

Résultats d'exécution obtenus :

Remplissage, contenu de la baignoire = 50 Remplissage, contenu de la baignoire = 100 Remplissage, contenu de la baignoire = 150 Remplissage, contenu de la baignoire = 200 Remplissage, contenu de la baignoire = 250 Remplissage, contenu de la baignoire = 300 Remplissage, contenu de la baignoire = 350 Remplissage, contenu de la baignoire = 400 Remplissage, contenu de la baignoire = 450 Remplissage, contenu de la baignoire = 500 Remplissage, contenu de la baignoire = 550 Remplissage, contenu de la baignoire = 600	Remplissage, contenu de la baignoire = 600 Fuite, contenu de la baignoire = 580 Remplissage, contenu de la baignoire = 630 Remplissage, contenu de la baignoire = 680 Remplissage, contenu de la baignoire = 730 Remplissage, contenu de la baignoire = 780 Remplissage, contenu de la baignoire = 830 Remplissage, contenu de la baignoire = 880 Remplissage, contenu de la baignoire = 930 Remplissage, contenu de la baignoire = 980 Remplissage, contenu de la baignoire = 1030 **Baignoire enfin pleine !**

---- operation complete.

Nous remarquons bien que le thread de remplissage Robinet a été privilégié dans ses exécutions, puisque dans l'exécution précédente le thread Baignoire-fuite n'a pu exécuter qu'un seul tour de boucle.

Alternons l'exécution de chaque thread :

Nous souhaitons maintenant que le programme alterne le remplissage de la baignoire avec la fuite d'une façon équilibrée : action-fuite/action-remplissage/action-fuite/action-remplissage/...

Nous utiliserons par exemple la méthode **yield ( )** qui cesse temporairement l'exécution d'un thread et donc laisse un autre thread prendre la main. Nous allons invoquer cette méthode yield dans chacune des boucles de chacun des deux threads **Robinet** et **Baignoire,** de telle sorte que lorsque le robinet s'interrompt c'est la baignoire qui fuit, puis quand celle-ci s'interrompt c'est le robinet qui reprend etc... :

Voici le code de la classe Robinet :

```
class Robinet extends Thread {
   public void debite(int quantite){
      for(int i=1; i<100; i++){
         if(Baignoire.contenu.volume < Baignoire.maximum){
            Baignoire.contenu.volume +=quantite;
            System.out.println("Remplissage, contenu de la baignoi
            Thread.yield();
         }
         else {
            System.out.println("Baignoire enfin pleine !");
            break;
         }
      }
   }

   public void run() {
      debite(50);
   }
}
```

Le corps de la méthode main de la classe principale lançant les actions de remplissage de la baignoire reste inchangé :

```
class RemplirThread {
   public static void main(String[] x){
      Baignoire UneBaignoire = new Baignoire();
      Robinet UnRobinet = new Robinet();
      UnRobinet.start();
      UneBaignoire.start();
   }
}
```

Annexe

Vocabulaire pratique : interprétation et compilation en java

Rappelons qu'un ordinateur ne sait exécuter que des programmes écrits en instructions machines compréhensibles par son processeur central. Java comme pascal, C etc... fait partie de la famille des langages évolués (ou langages de haut niveau) qui ne sont pas compréhensibles immédiatement par le processeur de l'ordinateur. Il est donc nécesaire d'effectuer une "**traduction**" d'un programme écrit en langage évolué afin que le processeur puisse l'exécuter.

Les deux voies utilisées pour exécuter un programme évolué sont la **compilation** ou l'**interprétation** :

Un compilateur du langage X pour un processeur P, est un logiciel qui traduit un programme source écrit en X en un programme cible écrit en instructions machines exécutables par le processeur P.

Un interpréteur du langage X pour le processeur P, est un logiciel qui ne produit pas de programme cible mais qui effectue lui-même immédiatement les opérations spécifiées par le programme source.

Un compromis assurant la portabilité d'un langage : une pseudo-machine

Lorsque le processeur P n'est pas une machine qui existe physiquement mais un logiciel simulant (ou interprétant) une machine on appelle cette machine pseudo-machine ou p-machine. Le programme source est alors traduit par le compilateur en instructions de la pseudo-machine et se dénomme pseudo-code. La p-machine standard peut ainsi être implantée dans n'importe quel ordinateur physique à travers un logiciel qui simule son comportement; un tel logiciel est appelé interpréteur de la p-machine.

La première p-machine d'un langage évolué a été construite pour le langage **pascal** assurant ainsi une large diffusion de ce langage et de sa version UCSD dans la mesure où le seul effort d'implémentation pour un ordinateur donné était d'écrire l'interpréteur de p-machine pascal, le reste de l'environnement de développement (éditeurs, compilateurs,...) étant écrit en pascal était fourni et fonctionnait dès que la p-machine était opérationnelle sur la plate-forme cible.

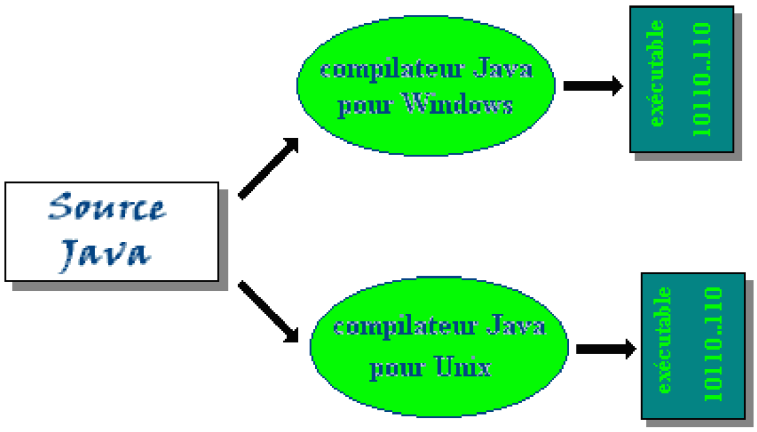
Donc dans le cas d'une p-machine le programme source est compilé, mais le programme cible est exécuté par l'interpréteur de la p-machine.

Beaucoup de langages possèdent pour une plate-forme fixée des interpréteurs ou des compilateurs, moins possèdent une p-machine, **Java est l'un de ces langages**. Nous décrivons ci-dessous le mode opératoire en Java.

Bytecode et Compilation native

Compilation native

La compilation native consiste en la traduction du source java (éventuellement préalablement traduit instantanément en code intermédiaire) en langage binaire exécutable sur la plate-forme concernée. Ce genre de compilation est équivalent à n'importe quelle compilation d'un langage dépendant de la plate-forme, l'**avantage est la rapidité d'exécution des instructions machines par le processeur central.**

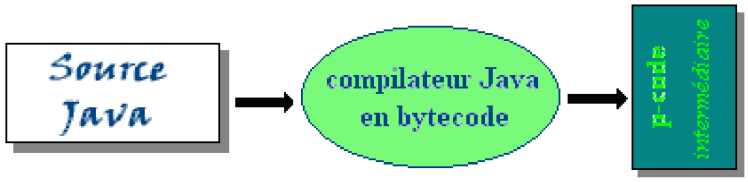


Programe source java : xxx.**java** (*portable*)
Programe exécutable sous windows : xxx.**exe** (*non portable*)

Bytecode

La compilation en bytecode (ou pseudo-code ou p-code ou code intermédiaire) est semblable à l'idée du p-code de N.Wirth pour obtenir un portage multi plate-formes du pascal. Le compilateur **Javac** traduit le programme source xxx.**java** en un code intermédiaire indépendant de toute machine physique et non exécutable directement, le fichier obtenu se dénomme xxx.**class**. Seule une p-machine (dénommée **machine virtuelle java**) est capable d'exécuter ce bytecode. Le bytecode est aussi dénommé **instructions virtuelles java**.

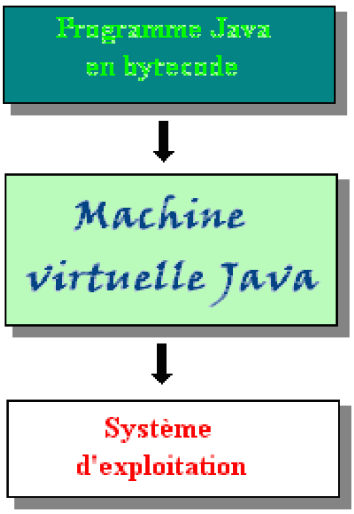
Figure : un programme source ***Exemple.java*** est traduit par le compilateur (dénommé **Javac**) en un programme cible écrit en bytecode nommé ***Exemple.class***



Exemple.java ----------> compilateur Javac ------> Exemple.class

La machine virtuelle Java

Une fois le programme source java traduit en bytecode, la machine virtuelle java se charge de l'exécuter sur la machine physique à travers son système d'exploitation (Windows, Unix, MacOs,...)



Inutile d'acheter une machine virtuelle java, tous les navigateurs internet modernes (en tout cas Internet explorer et Netscape) intègrent dans leur environnement une machine virtuelle java qui est donc installée sur votre machine physique et adaptée à votre système d'exploitation, dès que votre navigateur internet est opérationnel.

Fonctionnement élémentaire de la machine virtuelle Java

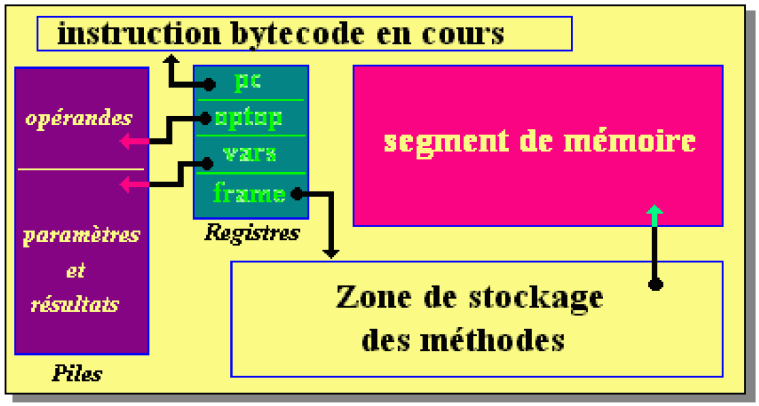
Une machine virtuelle Java contient 6 parties principales

- Un jeu d'instructions en pseudo-code
- Une pile d'exécution LIFO utilisée pour stocker les paramètres des méthodes et les résultats des méthodes
- Une file FIFO d'opérandes pour stocker les paramètres et les résultats des instructions du p-code (calculs)

- Un segment de mémoire dans lequel s'effectue l'allocation et la désallocation d'objets
- Une zone de stockage des méthodes contenant le p-code de chaque méthode et son environnement (tables des symboles,...)
- Un ensemble de registres (comme dans un processeur physique) servant à mémoriser les différents états de la machine et les informations utiles à l'exécution de l'instruction présente dans le registre instruction bytecode en cours.

Comme toute machine la machine virtuelle Java est fondée sur l'architecture de Von Neumann et elle exécute les instructions séquentiellement un à une.

Figure : un synoptique de la machine virtuelle Java



Les registres sont des mémoires 32 bits :

- **vars** : pointe dans la pile vers la première variable locale de la méthode en cours d'exécution.
- **pc** :compteur ordinal indiquant l'adresse de l'instruction de p-code en cours d'exécution.
- **optop** : sommet de pile des opérandes.
- **frame** : pointe sur le code et l'environnement de la méthode qui en cours d'exécution.

JIT , Hotspot

L'interprétation et l'exécution du bytecode ligne par ligne peut sembler prendre beaucoup de temps et faire paraître le langage Java comme "plus lent" par rapport à d'autres langages. Aussi dans un but d'optimisation de la vitesse d'exécution, des techniques palliatives sont employées dans les version récentes des machines virtuelles Java : la technique **Just-in-time** et la technique **Hotspot** sont les principales améliorations en terme de vitesse d'exécution.

***JIT** (Just-in-time)* est une technique de **traduction dynamique durant l'interprétation** que Sun utilise sous le vocable de **compilation en temps réel**. Il s'agit de rajouter à la machine virtuelle Java un compilateur optimiseur qui **recompile localement le bytecode** lors de son chargement et ensuite la machine virtuelle Java n'a plus qu'à faire exécuter des instructions machines de base. Cette technologie est disponible en interne sur les navigateurs de dernière génération.

On peut mentalement considérer qu'avec cette technique vous obtenez un programme java cible compilé en deux passages :

- le premier passage est dû à l'utilisation du compilateur **Javac** produisant du bytecode,
- le second passage étant le compilateur **JIT** lui-même qui optimise et traduit localement le bytecode en instructions du processeur de la plate-forme.

***Hotspot** est une amélioration de **JIT**.*

Un défaut dans la vitesse totale d'exécution d'un programme java sur une machine virtuelle Java équipée d'un compilateur **JIT** se retrouve dans le fait qu'une méthode qui n'est **utilisée qu'une seule fois se voit compilée puis ensuite exécutée**, les mesures de temps par rapport à sa seule interprétation montre que **dans cette éventualité l'interprétation est plus rapide**. La société ***Sun*** a donc mis au point une technologie palliative dénommée **Hotspot** qui a pour but de déterminer dynamiquement quel est le meilleur choix entre l'interprétation ou la compilation d'une méthode. **Hotspot** lancera la compilation des méthodes utilisées plusieurs fois et l'interprétation de celles qui ne le sont qu'une fois.

Bibliographie

Livres papier vendus par éditeur

Livres Java en français

Maurers, Baufeld, Müller & al, Grand livre Java 2, Micro Application, Paris (1999).
Brit schröter, Dossier spécial Java 2 référence, Micro Application, Paris (2000).
A.Tasso, Le livre de Java premier langage, Eyrolles, Paris (2001).
D.Acreman, S.Dupin, G.Moujeard, le programmeur JBuilder 3, Campus press, Paris (1999).
S.Holzner, Total Java, Eyrolles, Paris (2001).
E.&M.Niedermair, Programmation Java 2, Micro Application, Paris (2000).
E.&M.Niedermair, développement Java pour le web, Micro Application, Paris (2000).
M.Morisson & al, secrets d'experts Java, Simon & Scuster MacMillan, Paris (1996).
A.Mirecourt, PY Saumont, Java 2 Edition 2001, Osman-Eyrolles, Paris (2001).
C.Delannoy, Exercices en Java, Eyrolles, Paris (2001)
CS.Horstmann,G.Cornell, au coeur de Java2 notions fondamentales Vol1, Campus press, Paris (2001).
CS.Horstmann,G.Cornell, au coeur de Java2 fonctions avancées Vol2, Campus press, Paris (2002).
H.&P.Deitel, Java comment programmer, Ed. Reynald goulet inc., Canada (2002)
L.Fieux , codes en stock Java 2, Campus press, Paris (2002)
B.Aumaille, J2SE les fondamentaux de la programmation Java, Ed.ENI, Nantes (2002)
R.Chevallier, Java 2 l'intro, Campus press, Paris (2002)
R.Chevallier, Java 2 le tout en poche, Campus press, Paris (2003)
D.Flanagan, Java in a nutshell (trad en fr.), Ed O'REILLY, Paris (2001)
E.Friedmann, Java visuel pro, Ed. First interactive, Paris (2001)
B.Burd Java 2 pour les nuls Ed. First interactive, Paris (2002)
L.Lemay, R.Cadenhead, Java 2 le magnum, Campus press, Paris (2003).
J.Hubbard, structures de données en Java, Ediscience , Paris (2003)
J.Bougeault, Java la maîtrise, Tsoft-Eyrolles, Paris (2003).
D.Barnes & M.Kölling, conception objet en Java avec BlueJ,Pearson education, Paris (2003).

Pour élargir votre horizon Java et développement, un must :

M.Lai, UML et java, InterEditions, 3ème édition, Paris (2004).