

Le JAVA.

Introduction

JAVA est un langage développé par SUN et qui selon ses concepteurs est :

Simple

Orienté objets

Robuste et sûr

Indépendant des architectures matérielles

Multitâches.

Simple

Cette notion est relative mais par rapport au C++, JAVA est plus simple à utiliser. En particulier, il n'existe pas de pointeurs explicites et la gestion de la mémoire est transparente pour le programmeur.

Orienté objets

Un objet informatique est une entité munie de propriétés et de méthodes capables d'agir sur ses propriétés ou de réaliser certaines actions. Un objet contient des données et du code permettant de les manipuler. Dans JAVA, cette notion est poussée à l'extrême puisque dans ce langage tout est objet y compris le programme.

Robuste et sûr

Le typage des données est extrêmement strict. Aucune conversion de type implicite pouvant provoquer une perte de précision n'est possible. Comme les pointeurs ne sont pas accessibles les erreurs de gestion mémoire sont impossibles. Pour les applets, il est en principe impossible d'accéder aux ressources de la machine hôte. Enfin lors de l'exécution, on vérifie que le code généré par le compilateur n'a pas été altéré.

Indépendant des architectures matérielles

Le compilateur génère un code universel le « byte-code ». Un interpréteur spécifique à l'ordinateur hôte appelé « machine-virtuelle » permet l'exécution des programmes. La représentation des données étant indépendante de la machine qui exécute le code, les résultats des calculs sont indépendants de cette machine.

Dans les versions récentes des machines virtuelles, un compilateur génère un code directement exécutable sur la machine ce qui permet d'obtenir des performances comparables à celle d'un langage directement compilé.

Multitâches

JAVA permet l'exécution en apparence simultanée de plusieurs processus. En réalité on accorde de façon séquentielle un peu du temps processeur à chaque processus. On dit aussi multithread.

Les applets

JAVA permet de développer soit des *applications classiques* ayant accès à toutes les fonctionnalités du langage soit des *applets* qui sont des programmes destinés à être exécutés dans un navigateur WEB. Pour des raisons évidentes de sécurité, une applet ne peut pas manipuler la mémoire ni les fichiers de la machine hôte.

Lors du chargement d'une applet JAVA, le « byte-code » est vérifié par la machine virtuelle. En cas d'erreurs involontaires ou volontaires (pour essayer d'enfreindre les règles de sécurité) ce code est rejeté. Si le code est correct, la machine virtuelle l'exécute dans une fenêtre de dimensions invariables qui sont précisées dans la balise html qui demande le chargement de l'applet. Voici un exemple de code html (minimal) pour appeler une applet.

```
<html>
  <head>
    <title>calcomp</title>
  </head>
  <body>
    <applet code=calcomp.class name=calcomp width=450 height=510></applet>
  </body>
</html>
```

Le fait que dans une applet la fenêtre d'exécution soit fixe doit être pris en compte lors de l'écriture du code : il n'y aura pas de mise à l'échelle possible selon la résolution écran de la machine.

JAVA est un langage puissant et doté de possibilités graphiques faciles à mettre en œuvre ; il permet l'animation des pages html ce qui constitue la principale raison de son succès. Tous les navigateurs contiennent maintenant une machine virtuelle JAVA. Toutefois il se pose des problèmes de compatibilité. SUN a développé une nouvelle version 1.2 de JAVA qui n'est pas prise en compte directement par la machine virtuelle des navigateurs ; pour pouvoir l'utiliser, il faut charger sur les machines des « plug-in » qui modifient la machine virtuelle. Pour cette raison, je me limiterai à la version 1.0.2. et à la version 1.1.

1. Le langage

1.1 Les types de base

1.1.1. Les identificateurs

A chaque objet, variable, programme, on associe un nom.

Sont admis **tous les caractères alphanumériques**, le signe souligné _ et le caractère \$. Le premier caractère d'un identificateur ne peut pas être un chiffre.

JAVA est sensible à la casse des lettres.

Un identificateur ne doit pas appartenir à la liste des mots réservés du langage.

Abstract	Boolean	break	byte	byvalue
Case	Catch	char	class	const
Continue	Default	do	double	else
Extends	False	finally	float	for
Goto	If	implements	import	instanceof
Int	Interface	long	native	new
Null	Package	private	protected	public
Return	Short	static	super	switch
synchronized	This	threadsafe	throws	transient
True	Try	void	while	

1.1.2. Commentaires

Trois possibilités de délimiter les commentaires existent :

Le symbole // : ce qui suit ce symbole dans la ligne est considéré comme commentaire.

Les symboles /* et */ : ce qui est entre les symboles est considéré comme commentaire.

Les symboles /** et */ qui permettent avec un utilitaire de SUN de générer des commentaires au format html.

1.1.3. Variables

En JAVA, **il faut déclarer toutes les variables en précisant leur type**. On peut éventuellement ajouter des modificateurs. Ainsi déclarer une variable « final » revient à créer une constante car le compilateur refusera toutes les instructions modifiant la valeur de cette variable.

On peut effectuer une affectation ou assignation (donner une valeur) en déclarant une variable.

Lors de sa création une variable reçoit toujours une valeur par défaut. (0 pour les entiers, 0.0 pour les flottants, false pour les booléens, null pour les objets).

Le signe égal (=) est le symbole d'assignation.

Le point virgule (;) est le terminateur d'instruction.

```
int i;
double a,b,c;
final float PI = 3.14159f;
```

1.1.4. Les littéraux

Les littéraux définissent les valeurs qui sont utilisées par les programmes. Il en existe trois catégories les booléens, les nombres et les caractères.

1.1.4.1 Booléens

Deux valeurs possibles true (vrai) et false (faux).

1.1.4.2 Entiers

Il est possible d'exprimer les entiers en décimal, octal et hexadécimal.

```
int i = 10 ;//format décimal
int i = 010; //format octal : le nombre commence par un zéro.
int i = 0x10; /*format hexadécimal : le nombre commence par un zéro suivi d'un x (majuscule ou minuscule)*/
long k = 123L //forçage au type long par ajout d'un L (ou l) à la fin.
```

Il en existe 4 types qui diffèrent par la taille : **byte** (1 octet), **short** (2 octets), **int** (4 octets), **long** (8 octets).

Attention aux débordements :

```
short i = 20000,j=20000,k ; //k = i+j vaut -25536 !
```

Sur les 16 bits du type short, le bit de poids fort est le bit de signe et les 15 bits restants contiennent la valeur.

1.1.4.3 Flottants

Deux types de flottants existent : simple précision (float) ou double précision (double).

Diverses écritures sont possibles :

```
float x = 10.0f; // nombre suivi par F ou f (obligatoire)
double z = 1.0d; // nombre suivi par D ou d (facultatif)
double y = -1E-5,t=8.4E5,h=.25;
```

1.1.4.4 Caractères

La représentation d'un caractère est un caractère placé entre deux apostrophes ('). Les caractères de JAVA respectent le codage « Unicode » sur 2 octets.

```
char x = 'x';
char omega = (char)969 ;//affiche le caractère grec oméga ω.
```

Tableau des types de JAVA

Type	Taille	Genre	Gamme des valeurs possibles
boolean	8 bits	Booléen	true et false
byte	8 bits	Entier	-128 à +127
short	16 bits	Entier	-32768 à +32767
Int	32 bits	Entier	-2147483648 à 2147483647
long	64 bits	Entier	-9223372036854775808 à 9223372036854775807
float	32 bits	Flottant	1.401E-45 à 3.4028235E+38
double	64 bits	Flottant	5E-324 à 1.7976931348623157E+308
char	16 bits	caractère	Tout caractère unicode

1.1.4.5 Chaînes de caractères

Les chaînes de caractères sont formées par une succession de caractères entourés par des guillemets ("). Elles correspondent au type String qui est une classe à part entière dotée de constructeurs et de méthodes spécifiques.

```
String s = "ceci est une chaîne";
```

1.1.5. Les tableaux

1.1.5.1 Tableaux simples

Les tableaux sont déclarés en post-fixant soit le type des variables du tableau soit le nom de celui-ci avec des crochets.

```
int i[];  
int[] i;
```

Ce genre de déclarations est insuffisant pour pouvoir utiliser le tableau.

Il est nécessaire de préciser sa dimension (pour que le compilateur réserve la mémoire nécessaire)

Ceci est réalisé au moyen de l'instruction *new*.

```
int i[] = new int[10];
```

Cette instruction crée un tableau de 10 entiers dont les indices vont de 0 à 9.

Il est aussi possible d'initialiser un tableau lors de sa création :

```
int impairs[] = {1,3,5,7,9,11}; //tableau à 6 éléments.  
int x = impairs[2]; // x = 5
```

Le séparateur des valeurs est la virgule.

1.1.5.2 Tableaux multidimensionnels

La syntaxe est la même que pour les tableaux à une dimension :

```
int tab[][] = new int[4][3]; //indices 0, 1, 2, 3 et 0, 1, 2  
float mat[][] = new float[3][3];
```

Comme pour les tableaux à une dimension, Il est possible d'initialiser un tableau lors de sa création :

```
int carre = {{11,12,13},{21,22,23},{31,32,33}};  
int x = carre[1][2]; //x = 23
```

Dans un bloc de valeurs, le séparateur est la virgule. Les blocs sont séparés par des virgules

1.1.6. Portée des variables

1.1.6.1 Blocs d'instructions.

Un bloc est un ensemble d'instructions contenues entre deux accolades { }

Il est possible d'imbriquer des blocs.

1.1.6.2 Portée des variables

L'emplacement de la déclaration de la variable définit sa zone de visibilité. Une variable est visible dans le bloc dans lequel elle est définie.

```
class test{ //début du bloc test
  int k=5;
  void methode1()
  { // début de methode1
    float x=4;
    ...
  } //fin de methode1.

  void methode2()
  { // début de methode2
    double z;
    int k=8;
    ...
  } //fin de methode2.

  void methode3()
  { // début de methode3
    double y;
    ...
  } //fin de methode3.
} //fin de test
```

Dans le programme ci-dessus, la variable x n'est visible que dans *methode1*. La variable z n'est visible que dans *methode2*.

Par contre k est visible dans *methode1* et *methode3* avec la valeur 5. Mais dans *methode2* l'affectation `int k = 8` masque la variable k définie dans test. Dans le bloc *methode2*, la valeur initiale de k est 8.

1.1.7 Conversions de types

Les conversions de type doivent être faites explicitement en JAVA.

On encadre l'identificateur du type désiré par des parenthèses ().

```
char c; double d; int i=64;
c = (char)i; //conversion de la valeur de i en caractère
i = (int) d; //effectue la troncature de d
```

Les conversions de type peuvent entraîner des pertes de précision.

De vers	byte	short	int	long	float	double	char
byte	*	*	*	*	*	*	*
short		*	*	*	*	*	*
int			*	*	perte	*	*
long				*	perte	perte	*
float					*	*	*
double						*	*
char							*

Par exemple à la suite de l'instruction `char c = (char)64.5f`; c est égal à @

1.2. Les opérateurs

1.2.1 Opérateurs numériques

Ils sont classés selon le nombre d'arguments nécessaires en opérateurs unaires et binaires.

1.2.1.1 Opérateurs unaires

Opérateur	Action	Exemple
-	Négation	$i = -k$;
++	Incrément de 1	$i++$; // $i = i + 1$
--	Décrément de 1	$i--$; // $i = i - 1$

1.2.1.2 Opérateurs binaires

Opérateur	Action	Exemple
+	Addition	$c = a + b$;
+=		$i += j$; // $i = i + j$
-	Soustraction	$c = a - b$;
-=		$i -= 2$; // $i = i - 2$
*	Multiplication	$c = a * b$;
*=		$c *= a$; // $c = c * a$
/	Division (entière si arguments entiers)	$c = a / b$;
/=		$c /= a$; // $c = c / a$
%	Modulo	$i = j \% n$;
%=		$i \% = 4$; // $i = i \% 4$

1.2.2 Opérateurs relationnels

Opérateur	Action	Exemple
<	Plus petit que	$a < b$;
>	Plus grand que	$a > b$;
<=	Plus petit ou égal à	$a <= b$;
>=	Plus grand ou égal à	$a >= b$;
==	Egal à	$a == b$;
!=	Différent de	$a != b$;

Noter la différence entre l'assignation (=) et l'égalité (==).

1.2.3 Opérateurs logiques

Opérateur	Action	Exemple
!	Négation (Complément)	<code>c = !c ;</code>
&	ET	<code>a & b;</code>
	OU	<code>a (b>3) ;</code>
^	OU exclusif	<code>a ^ b;</code>
~	NON logique	<code>a = ~b;</code>
&&	ET évalué	<code>a && b && c ;</code>
	OU évalué	<code>a b c ;</code>
!=	Complément assigné	<code>c != c ;</code>
&=	ET assigné	<code>a &= b ;//a = a & b</code>
=	OU assigné	<code>a = b ;//a = a b</code>

Si on utilise les opérateurs évalués && et ||, l'évaluation de l'expression cesse dès qu'une certitude est acquise (premier terme false dans un &&, ou premier terme true dans un ||).

1.2.4 Opérateurs de manipulation des binaires

Opérateur	Action	Exemple
<<	Décalage à gauche.	<code>i << k ;</code> décaler i vers la gauche de k bits.
>>	Décalage à droite signé.	<code>i >> k ;</code> décaler i vers la droite de k bits avec son signe
>>>	Décalage à droite non signé	<code>i >>> k ;</code> décaler i vers la droite de k bits sans son signe

Ces opérations de manipulation sur les binaires correspondent à des opérations arithmétiques sur des entiers.

`~i; =>(i-1)-1`
`i >> k => i / 2k si i > 0`
`i << k => i * 2k`

1.2.5 Précédence des opérateurs

Dans le tableau les opérations sont classées par ordre de priorité.

++	--	!	~
*	/	%	
+	-		
<<	>>	>>>	
<	>	<=	>=
==	!=		
&			
^			
&&			

Il est toujours possible de modifier cet ordre en utilisant des parenthèses.

2. Les structures de contrôle

2.1. Blocs d'instructions

Un bloc est un ensemble d'instructions contenues entre deux accolades `{ }`. Il définit la visibilité (portée) des variables qui y sont déclarées.

Il est possible d'imbriquer des blocs.

2.2. Exécution conditionnelle

Il existe en JAVA deux structures :

Si alors sinon (*if_then_else*)

Au cas ou (*switch*)

2.2.1 Structure *if_then_else*

Sous sa forme complète, cette instruction est codée :

```
if (expression) blocvrai; else blocfaux;
```

L'expression du **test qui doit être écrit entre deux parenthèses ()** est une expression booléenne.

Si expression est vraie, alors on exécute les instructions contenues dans *blocvrai* sinon on exécute celles contenues dans *blocfaux*.

```
if (i==3) k = 1; else k = -1;
```

Il est possible d'omettre le *else*. Si expression est fausse, alors aucun code n'est exécuté.

Il est possible d'imbriquer des structures *if_then_else*.

```
if (a == 1)
    st = "a vaut 1";
else if (a == 2)
    st = "a vaut 2";
else st = "a ne vaut pas 1 ou 2";
```

La branche *else* est rattachée au dernier *if* rencontré.

En cas de doute, il faut utiliser des accolades.

Exemple :

```
if (a == 1)
    if (b == 1) inst1; else inst2;
```

inst2 n'est exécutée que si a vaut 1 et si b est différent de 1.

Il est préférable d'écrire :

```
if (a == 1){
    if (b == 1) inst1; else inst2;}
```

2.2.2 L'opérateur ternaire

C'est un opérateur spécial qui autorise un test simple sans utiliser l'instruction *if* :

```
variable = condition ? siTestVrai : si TestFaux
```

variable prend la valeur *siTestVrai* si le test booléen *condition* est vrai et la valeur *siTestFaux* dans le cas contraire.

Exemple :

```
int a = 1, b=3;
int mini = (a<b) ? a : b; //les parenthèses sont facultatives
```

Ce code est équivalent à :

```
int a = 1, b = 3;
int mini ;
if (a < b) then mini = a ; else mini = b;
```

2.2.3 Structure switch

En cas de tests multiples, il est préférable d'utiliser l'instruction *switch* plutôt qu'une série de *if* imbriqués. Cette instruction se comporte comme un aiguillage qui exécute des instructions différentes (*case*) selon le résultat d'une expression logique. A l'entrée dans la structure, cette expression est évaluée et son résultat est comparé à chaque valeur des instructions *case*. En cas d'égalité, les instructions de cette clause sont exécutées jusqu'à la rencontre d'une instruction *break* qui entraîne la sortie de la structure *switch*. La clause (optionnelle) *default* est exécutée lorsque la valeur de l'expression d'aiguillage n'est égale à aucun des cas. D'un point de vue logique, les clauses *case* sont considérées comme des étiquettes (label) : elles sont donc suivies du caractère « : ». (cf § 2.3.4)

Le type de la variable d'une instruction case doit être char, byte, short ou int.

Structure :

```
switch (variable){ //début du bloc switch
  case cas1: //cas1 = char, byte, short ou int
    instruc11;
    instruc12;
    break ; //si ce break est omis, on exécute aussi instruc21 et instruc22
  case cas2 :
    instruc21;
    instruc22;
    break;
  default :
    instruc_par_defaut;
} // fin du bloc switch
```

Exemple :

```
int erreur ;
...
switch (erreur){
  case 1 : afficher("Argument de fonction non valide");
    erreur=0; break;
  case 2 : afficher("Syntaxe incorrecte");
    erreur=0; break;
  case 3 : afficher("Division par zéro");
    erreur=0; break;
  default : afficher("Erreur inconnue");}
```

Selon la valeur de *erreur*, on sélectionne l'argument de la procédure *afficher*().

2.3 Exécutions itératives

Ces structures permettent une exécution répétitive du code (boucle). L'exécution cesse quand une condition de test n'est plus remplie. Dans JAVA, il existe trois structures itératives, while, do...while et for.

2.3.1 Boucles while

On exécute un bloc d'instruction **tant que** une condition est vraie.

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

Déroulement :

A l'entrée dans la structure, on évalue le booléen (*condition*). Si le test est vrai, on exécute l'instruction qui suit (ou le bloc). On réévalue la condition. Si elle est toujours vraie, on exécute à nouveau le corps de la boucle et ainsi de suite jusqu'à la condition devienne fausse. On saute alors à l'instruction qui suit le corps de la boucle. Si le test est faux au départ, le contenu de la boucle n'est jamais exécuté.

L'une des instructions du corps de la boucle doit faire évoluer la valeur de condition sinon on obtient une boucle sans fin.

Exemple :

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

A la sortie de la boucle, somme contient la somme des 100 premiers entiers.

Attention de ne pas mettre de point virgule après la condition.

Quand on fait cette erreur, non décelée par le compilateur (car la syntaxe est correcte) le code qui suit le point virgule n'est pas considéré comme faisant parti de la structure while !

Exemple :

```
int i = 100, j = 20;  
while (j <= i);  
{ ...  
    j++;}
```

La valeur finale de j est 21.

2.3.2 Boucle do_while

Cette structure est très proche de la structure while. Sa syntaxe est :

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

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

Exemple :

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

A la sortie de la boucle, la variable somme contient la somme des 100 premiers entiers.

2.3.3 Boucles for

La structure de la boucle for est la suivante :

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

Si le corps de la boucle contient plusieurs instructions, on utilise des accolades pour définir le bloc à itérer.

exp1 initialise le compteur (après une déclaration éventuelle auquel cas la portée de la variable compteur est limitée à la boucle)

exp2 est une condition de test qui est évaluée à chaque passage. La boucle est exécutée tant que *exp2* est vraie.

Après exécution du contenu de la boucle, *exp3* est exécutée. Cette expression est souvent l'incrémentement du compteur mais ce n'est pas obligatoire.

Une boucle for est alors équivalente à la structure while suivante :

```
initialisation_compteur;
while (condition){
    instructions ;
    incrementation ;}
```

On utilise en général la boucle *for* quand le nombre d'itérations est connu a priori.

Exemple :

```
int somme = 0;
for (int i=0; i<=100; i++) //la portée de i est limitée au bloc for
    somme+= i;
```

mais la structure suivante est également correcte :

```
int somme = 0;
for (int i=0; i<=100;){
    somme += i;
    i++ ;}
```

Remarques :

* *exp1* et *exp3* peuvent contenir plusieurs instructions qui sont alors séparées par des virgules.

Exemple :

```
for (int i=0,j=10; i<=100; i++,j--)
```

```
instructions;
```

* La boucle : `for (; ;) {instructions;}` correspond à la boucle : `while (true){instructions;}`

Il est possible d'*imbriquer* toutes les structures itératives. Il faut alors veiller à la portée des variables locales.

2.3.4 Etiquettes

Il est possible d'identifier une instruction ou un bloc avec une étiquette (**label**). L'étiquette est formée d'un nom arbitraire suivi du caractère « deux points ». Cette possibilité est utilisée avec les instructions de rupture *break* et *continue*.

2.3.5 Break

Cette instruction (déjà rencontrée dans switch) est aussi utilisable dans les trois structures itératives. Elle sert à interrompre le flot des instructions et à sortir du bloc courant sans exécuter les instructions qui suivent le break. Cette instruction permet de quitter « proprement » une structure itérative.

Exemple :

On cherche si un tableau `tab[]` de 100 entiers contient la valeur 5.

```
int valeur = 5 ;
boolean ok = false ;
for (int i = 0 ; i<100; i++){
    if (tab[i]==valeur){
        ok = true;
        break;}}
if (ok)...
```

On quitte la boucle dès que la valeur cherchée est trouvée.

On cherche maintenant si un tableau à deux dimensions `tab[][]` de 100*50 entiers contient la valeur 5 et on veut quitter la boucle dès que cette valeur est trouvée. Le code suivant :

```
int valeur = 5 ;
boolean ok = false ;
for (int i = 0 ; i<100; i++){
    for (int j = 0 ; j<50; j++){
        if (tab[i][j] == valeur){
            ok = true;
            break;}}
if (ok) ...
```

ne fonctionne pas de la manière souhaitée car le `break` fait quitter seulement la boucle d'indice `j`.

Il faut utiliser un `break label`. Quand cette instruction est rencontrée, on quitte le bloc nommé par l'étiquette `label`.

```
int valeur = 5 ;
boolean ok = false ;
bcl1 : //label
for (int i = 0 ; i<100; i++){
    for (int j = 0 ; j<50; j++){
        if (tab[i][j] == valeur){
            ok = true;
            break bcl1;}}
if (ok) ...
```

2.3.6 Continue

Cette instruction modifie aussi le déroulement normal d'une boucle. Elle permet de sauter les instructions qui suivent `continue` et de redémarrer le flot d'instructions au niveau de l'évaluation de la condition de la boucle.

Contrairement à l'instruction `break` qui fait **quitter** la boucle, on saute les instructions qui suivent `continue` puis on **continue** l'exécution de l'itération.

L'instruction `continue` peut aussi se combiner avec une étiquette.

```
bcl1 : //label
for (int i = 0 ; i<100; i++){
    for (int j = 0 ; j<50; j++){
        ...
        if (j == 15) continue; // retour début boucle d'indice j
        ...
        if (i == 5) continue bcl1; // retour début boucle d'indice i
    ... } }
```

2.3.7 Return

Cette instruction termine **immédiatement** l'exécution des méthodes et procédures. On peut donc considérer que c'est aussi une instruction de contrôle.

3. Les objets en JAVA

Les concepteurs de langages cherchent les moyens de faciliter la création et la maintenance du code. Une méthode souvent retenue consiste à structurer le langage au moyen de « procédures » qui sont des blocs de code effectuant une action bien précise sur des données du programme. Il est possible de regrouper ces procédures en bibliothèques utilisables par d'autres programmeurs. Cette façon de faire est efficace mais présente l'inconvénient de séparer le code et les données et d'être sensible aux effets de bord. L'idée de base des langages objets est de regrouper données et code en une même entité l'objet. Cette réunion données-code se nomme « encapsulation ».

Quelques définitions relatives aux classes :

Classe : c'est le moule qui permet de fabriquer les objets.

Objet : c'est une réalisation concrète et utilisable de la classe. En jargon objet, on dit que c'est une « instance » de la classe.

Méthodes : ce sont des procédures qui réalisent les actions que l'on peut effectuer sur l'objet.

Message : c'est l'appel d'une méthode de l'objet.

3.1 Classes et objets dans JAVA

Nous allons illustrer la description des classes par un exemple simple. Nous allons créer la classe « rectangle » qui affiche et manipule des rectangles.

3.1.1 Déclaration de classe

La syntaxe de la déclaration d'une classe est :

```
class rectangle {  
    ...  
}
```

3.1.2 Variables et méthodes d'instance

Les variables d'instance sont les données propres à chaque objet (instance de la classe). Les variables relatives à deux objets différents sont physiquement distinctes : elles occupent des cases mémoires différentes.

Pour définir un rectangle, il faut connaître la position d'un sommet (supérieur gauche par exemple) sa largeur et sa hauteur.

```
class rectangle {  
    int orx,ory,large,haut;  
    ...  
}
```

Les méthodes vont déterminer le comportement des objets *rectangle*.

La syntaxe de la déclaration d'une méthode est :

Type du résultat produit par la méthode ou *void* si elle ne produit pas de résultat ;

Le nom de la méthode ;

Le type et le nom des arguments placés entre parenthèses (). Les parenthèses même vides sont obligatoires.

Le [mode de passage des arguments](#) des méthodes est précisé dans une note.

Un bloc d'instruction qui constitue la méthode. Sauf pour les méthodes de type void, le bloc de code est terminé par *return le_resultat*.

La première méthode calcule la surface du rectangle :

```
int surface()
{ int surf=large*haut;
  return surf; }
```

La seconde dessine le rectangle en noir. On utilise ici des instructions de java.awt qui est un ensemble de classes utilitaires livrées avec JAVA. L'instruction "g.setColor(Color.black)" signifie que la couleur du pinceau pour écrire dans l'objet graphique g est la couleur *Color.black*.

```
void dessine()
{ g.setColor(Color.black);
  g.drawRect(orm, ory, large, haut); }
```

L'objet graphique g doit être défini comme variable pour la classe par l'instruction « Graphics g ; ».

3.1.3 Constructeurs

C'est une méthode particulière qui indique comment initialiser la classe.

Cette méthode *doit* porter le nom de la classe.

```
rectangle(Graphics appG, int x, int y, int l, int h)
{ g= appG;
  orm = x; ory = y; large = l; haut = h; }
```

Pour chaque nouvel objet rectangle, ce constructeur initialise ses variables d'instances avec les valeurs des paramètres passées au constructeur : appG, x, y, l et h.

Dans une classe, la présence des constructeurs est facultative.

Voici le code complet de notre classe *rectangle*.

```
class rectangle{
  int orm,ory,large,haut; //variables d'instance
  Graphics g;

  rectangle (Graphics appG, int x, int y,int l, int h) //constructeur
  { g=appG;
    orm=x; ory = y; large = l; haut = h; }

  int surface() //méthodes
  { int surf=large*haut;
    return surf; }

  void dessine()
  { g.setColor(Clou.black);
    g.drawRect(orm, ory, large, haut); }
}
```

3.1.4 Appels des méthodes

L'invocation d'une méthode se fait en donnant le nom de l'instance puis celui de la méthode en précisant les arguments (éventuels) de celle-ci.

Si la liste des arguments est vide, la présence des parenthèses est quand même nécessaire.

Exemple :

Pour dessiner un rectangle de nom rec, le message d'appel à la méthode sera : `rec.dessine()` ;

3.1.5 Utilisation d'une classe

Pour pouvoir utiliser la classe rectangle, nous allons créer une applet dont le détail du code sera explicité ultérieurement.

```
import java.applet.*;
import java.awt.*;

public class testrect extends Applet
{ rectangle r1,r2; // note 1

public init()
{ setBackground(Color.ligthGray);
  r1=new rectangle(getGraphics(),10,20,100,50); //note 2
  r2=new rectangle(getGraphics(),10,80,100,60)}; //note 3

public void paint(Graphics g)
{ int x=r1.surface(); //note 4
  g.drawString(""+x,250,100); //note 5
  r1.dessine(); //note 6
  r2.dessine();
}
```

Ce programme (applet) est une classe qui ne possède pas de constructeur et deux méthodes que l'on retrouve dans toute les applets. Au démarrage de l'applet, le navigateur appelle la méthode *init()* puis la méthode *paint()* qui dessine dans la fenêtre octroyée par le navigateur à l'applet.

Note 1 : on définit deux instances de la classe rectangle r1 et r2 ;

Note 2 : on crée effectivement avec l'instruction *new* l'instance de nom r1 de la classe. Le compilateur réserve la mémoire nécessaire pour contenir les informations relatives à r1. L'instruction *getGraphics()* permet de récupérer l'objet graphique de l'applet.

Note 3 : on crée effectivement la nouvelle instance r2 de la classe.

Note 4 : on crée la variable x à laquelle est affectée la valeur de la surface du rectangle r1.

L'exécution de l'instruction : *r1.surface()* à la place de : *int x = r1.surface()* provoque aussi un appel à la méthode *surface* mais la valeur calculée n'est pas récupérée.

Note 5 : cette instruction entraîne l'affichage au point de coordonnées x = 250, y =100 de la valeur de x.

Note 6 : appel de la méthode *dessine()* qui provoque le dessin du rectangle r1 dans la fenêtre de l'applet ;

Cette applet sera appelée par une page html dont le code (minimal) sera :

```
<html>
  <head>
    <title>testrect</title>
  </head>
  <body>
    <applet code=testrect.class, width=320, height=240 ></applet>
  </body>
</html>
```

3.1.6 Constructeurs multiples

Le concepteur de la classe rectangle se rend compte qu'en plus des rectangles, il devra manipuler aussi des rectangles pleins de diverses couleurs. Au lieu de créer une nouvelle classe, il est possible en JAVA de créer un nouveau constructeur. Il aura le même nom que le constructeur initial mais devra différer de celui-ci soit par le nombre de ses arguments, soit par leur type.

Voici une forme possible de ce nouveau constructeur (qui possède l'argument supplémentaire *couleur* du type *Color* qui est un des nombreux types de java.awt).

```
rectangle (Graphics appG,int x, int y, int l, int h, Color couleur)
{ g = appG;
  orx = x; ory =y; large = l; haut = h;
  plein = true; col =couleur;}
```

Il faut ajouter deux variables d'instance à la classe rectangle : la variable *col* qui est de type *Color* et le booléen *plein* qui sera utilisé de façon interne à la classe afin de distinguer les rectangles simples des rectangles pleins. On pourra modifier la méthode *dessine()* pour prendre en compte les deux types de rectangles possibles.

```
public void dessine()
{ if (plein){
  g.setColor(col); g.fillRect(ors,ory,large,haut);}
  g.setColor(Color.black);
  g.drawRect(ors,ory,large,haut);}
```

Il est aussi possible de modifier la méthode *surface* pour lui faire afficher dans le rectangle la valeur de sa surface.

```
public int surface()
{ int sur = large*haut;
  this.dessine(); // ou dessine( )
  g.drawString(""+sur,ors+10,ory+20);
  return sur;}
```

Dans certains cas, il peut y avoir ambiguïté sur l'instance à prendre en compte. Le mot clé *this* permet de préciser que la méthode doit être appliquée à l'instance en cours d'utilisation.

3.1.7 Modificateurs et portée des variables d'instance

Les variables d'instance sont accessibles directement par toutes les méthodes de la classe. Pour respecter l'encapsulation, il ne faut pas que les données d'un objet puissent être modifiées directement depuis l'extérieur de celui-ci. Pour y parvenir, on peut limiter la portée des variables de classe (et des méthodes) au moyen de modificateurs. Dans le cas présent, les modificateurs utilisables sont *private* qui limite la portée des variables à la classe et *public* qui l'étend à toutes les classes.

Une variable de classe déclarée *public* est accessible depuis l'extérieur de la classe en préfixant son nom par celui de l'objet. Par défaut, variables et méthodes sont *public*.

3.1.8 Forme finale de l'exemple

Comme exemple de l'utilisation des modificateurs, la variable `col` est déclarée *public* : elle est accessible à partir de la classe `testrect` en utilisant la syntaxe : `Nom_de_l'instance.col = valeur`. (noter le point entre le nom de l'objet et la variable).

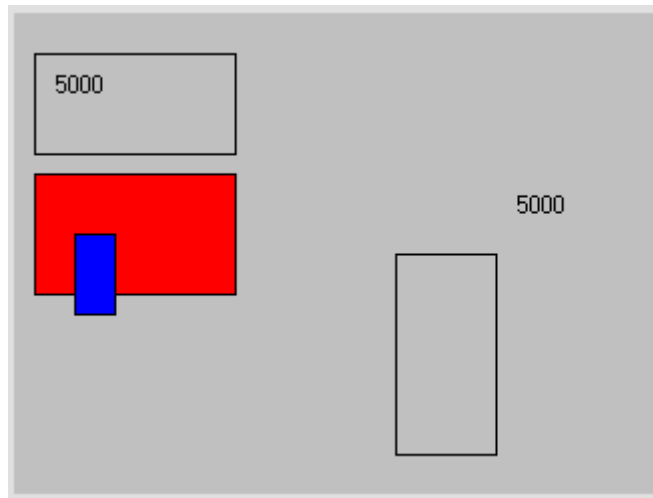
Par contre, les autres variables étant déclarées *private* sont inaccessibles depuis `testrect` : la ligne `r1.orx = 50` ; placée dans `paint` provoque une erreur de compilation puisque « `orx` » a été déclarée *private*.

```
import java.applet.*;
import java.awt.*;
class rectangle1
{ private Graphics g;
  private int orx,ory,large,haut;
  private static float echelle=1.0f;
  public Color col;
  private boolean plein;
rectangle1(Graphics appG,int x, int y, int l, int h)//constructeur 1
{ g=appG;
  orx=x; ory=y; large=l; haut=h;
  plein=false;}
rectangle1(Graphics appG,int x, int y, int l, int h,Color couleur) //constructeur 2
{ g=appG;
  orx=x; ory=y; large=l; haut=h;
  plein=true; col=couleur;}
public void dessine()
{ if (plein){
  g.setColor(col); g.fillRect(orx,ory,large,haut);}
  g.setColor(Color.black);
  g.drawRect(orx,ory,large,haut);}
public void tourne()
{ int temp;
  temp=large; large=haut; haut=temp;}
public void bouge(int dx, int dy)
{ orx += dx; ory += dy;}
public void change(int l, int h)
{ large = l; this.haut = h;}//this est facultatif
public int surface()
{ int sur=(int)(large*haut*echelle);
  this.dessine();
  g.drawString(""+sur,orx+10,ory+20);
  return sur;}
}
//*****
public class testrect1 extends Applet
{ rectangle1 r1,r2;
public void init()
{ setBackground(Color.lightGray);
  r1= new rectangle1(getGraphics(),10,20,100,50);
  r2= new rectangle1(getGraphics(),10,80,100,60,Color.red);}
public void paint(Graphics g)
{ int x = r1.surface(); g.drawString(""+x,250,100);
  r1.dessine();
  r2.dessine();
  r1.tourne();
  r1.bouge(180,100);
  r1.dessine();
  r2.bouge(20,30);
  r2.col = Color.blue;
  r2.change(20,40);
```

```

    r2.dessine();}
}

```



3.1.9 Surcharge des méthodes

De la même façon qu'il est possible de définir plusieurs constructeurs, il est possible de définir deux méthodes ayant le même nom à la condition que leur nombre d'arguments ou que les types des arguments diffèrent. Cette possibilité s'appelle la surcharge des méthodes.

Exemple : Afin de créer une homothétie, on peut surcharger la méthode :

```

public void change(int l, int h)
{   large = l; haut = h;}

```

de la manière suivante :

```

public void change(float k)
{   large = (int)(large*k).
    haut = (int)(haut*k);}

```

3.1.10 Variables et méthodes de classe

3.1.10.1 Variables de classe (statiques)

Dans la classe rectangle, on pourrait définir la variable `nombre_d_or`. Cette variable aura la même valeur pour toutes les instances de la classe. Il est donc inutile de la dupliquer dans toutes les instances. Ceci est obtenu en déclarant cette variable *static*. Les variables statiques, déclarées *static* sont dites variables de classe.

Les variables de classe sont initialisées une fois pour toute lors du chargement de celle-ci.

Dans l'exemple précédent, la variable `Graphics g` ayant la même valeur pour toutes les instances peut être déclarée *static*.

3.1.10.2 Portée des variables de classe

Elles sont utilisables directement dans toutes les méthodes de la classe.

Elles sont aussi utilisables depuis l'extérieur de la classe. Leur nom doit alors être préfixé par le nom de la classe (et pas par le nom d'une instance !)

3.1.10.3 Méthodes de classes

Ce sont des méthodes destinées à agir sur la classe plutôt que sur les instances. On doit les déclarer *static*. Pour accéder à une méthode statique depuis l'extérieur de la classe, il faut préfixer le nom de la méthode par le nom de la classe ou par le nom d'une instance de la classe.

Pour améliorer la lisibilité du code il est conseillé de **préfixer le nom de la méthode par le nom de la classe**.

Dans la suite, nous utiliserons uniquement cette écriture.

3.1.10.4 Exemple

L'exemple suivant donne une manière possible d'implémenter une classe permettant de manipuler les nombres complexes.

```
class comp
{ double a,b; //variables d'instance
  final static double PI = Math.PI; //variable de classe

  comp(double inita,double initb) //constructeur
  {a=inita; b=initb;}

  static double norme(comp x) //méthodes de classe
  { return Math.sqrt(x.a*x.a + x.b*x.b);}

  static double phase(comp x)
  { return Math.atan2(x.b,x.a)*180/PI;}

  static comp somme(comp x, comp y)
  { comp s=new comp(0,0);
    s.a = x.a + y.a; s.b = x.b + y.b;
    return s;}

  static comp produit(comp x, comp y)
  { comp p=new comp(0,0);
    p.a = x.a*y.a - x.b*y.b; p.b = x.b*y.a + y.b*x.a;
    return p;}

  static comp quotient(comp x, comp y)
  { comp q=new comp(0,0);
    double n2=y.a*y.a + y.b*y.b;
    q.a = (x.a*y.a + x.b*y.b)/n2; q.b = (x.b*y.a - y.b*x.a)/n2;
    return q;}
}
```

Pour utiliser cette classe, on pourra par exemple écrire :

```
comp a = new comp (3,5); // déclaration et initialisation
comp b = new comp (2,-4);
comp c= new comp (0,0);
c = comp.produit (a,b); // appel d'une méthode
```

Remarques :

Les méthodes sont publiques par défaut.

Math.sqrt est un appel à la méthode statique sqrt (racine carrée) de la classe « Math » de JAVA.

3.1.11 Comparaison d'objets

3.1.11.1 Références des objets

Après l'exécution du code suivant qui concerne des littéraux :

```
int i = 2 ;  
int j = i ;  
i = 4;
```

la variable `i` vaut 4 et la variable `j` vaut 2. Quand on travaille sur des littéraux, toute déclaration de variable réserve pour celle-ci une zone mémoire de la taille idoine (par exemple 4 octets pour un entier de type `int`). Le code `j = i` recopie la valeur de la zone mémoire "`i`" dans la zone mémoire "`j`".

Il en va différemment avec des objets. Après l'exécution du code suivant :

```
rectangle r1, r2;  
r1 = new rectangle(getGraphics(),10,20,100,50);  
r2 = r1;  
r1.change(60,30);
```

la largeur du rectangle `r2` vaut aussi 60 et sa hauteur 30 car `r2` et `r1` correspondent en fait au même objet. Toute modification des variables de l'un modifie les variables de l'autre. On dit que `r1` et `r2` pointent sur un même objet.

Quand on déclare un objet, on récupère l'adresse d'une case mémoire qui est en général l'adresse de la première case mémoire de la zone réservée pour contenir toutes les données relatives à l'objet. Toute manipulation de cet objet fait référence à cette adresse (pointeur implicite).

L'instruction `r2 = r1` recopie seulement l'adresse du pointeur vers `r1` dans le pointeur vers `r2`. Pour créer une nouvelle instance indépendante d'un objet, il faut utiliser son constructeur.

3.1.11.2 Comparaison des objets

La comparaison de deux instances d'un objet (avec `==`) s'effectue sur les références de ces instances. Après exécution de :

```
Boolean ok = false ;  
rectangle r1, r2;  
r1 = new rectangle(getGraphics(),10,20,100,50);  
r2 = new rectangle(getGraphics(),10,20,100,50);  
if (r1 == r2) ok = true;
```

`ok` reste "false" car les références de `r1` et de `r2` sont différentes.

La comparaison de deux instances d'une classe nécessite l'écriture d'une méthode spécifique dans laquelle on compare la valeur de chacune des variables définies par le constructeur de l'objet..

3.2 Héritage

C'est un processus qui permet d'ajouter des fonctionnalités à une classe sans avoir à réécrire tout le code de cette classe. La nouvelle classe hérite de toutes les données et méthodes de la classe dont elle est issue. Pour signifier que la nouvelle classe dérive d'une classe mère, on ajoute à la fin de sa déclaration le modificateur *extends* suivi du nom de la classe mère. Les classes filles n'héritent pas directement des constructeurs du parent : On doit, avec le mot clé *super*, faire appel au constructeur de la classe mère puis initialiser les variables spécifiques à la nouvelle classe. Si la première instruction du constructeur d'une sous-classe n'est pas *super* (référence au constructeur de la classe mère), un appel *super()* est exécuté automatiquement par le compilateur. Il faut alors qu'il existe un constructeur sans paramètre dans la classe mère.

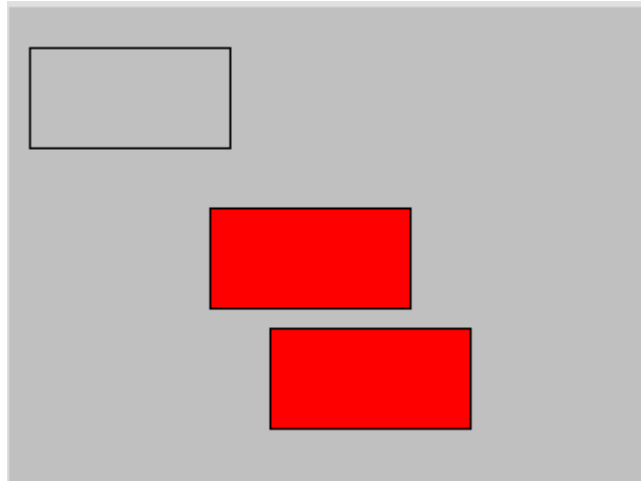
Par exemple pour tracer des rectangles pleins, il est possible d'utiliser une approche différente de celle qui a déjà été examinée. On peut par exemple créer une nouvelle classe *rectplein* qui va hériter de la classe mère *rectangle*. Pour que la classe *rectplein* puisse accéder aux données de *rectangle*, il faudrait que celles-ci soient publiques ce qui est contraire au principe de l'encapsulation. La solution est de définir les variables de classe avec le modificateur *protected*. Les membres d'une classe (données, méthodes) protégés par "protected" sont accessibles à partir des classes dérivées mais pas des autres classes. La classe *rectplein* hérite de toutes les méthodes de *rectangle* qui ne sont pas redéfinies dans *rectplein*. La méthode *bouge()* est celle de *rectangle* ; par contre comme la méthode *dessine* a été redéfinie dans la classe *rectplein* c'est elle qui est utilisée pour les objets de ce nouveau type.

```
import java.applet.*;
import java.awt.*;
class rectangle
{ protected static Graphics g;
  protected int orx,ory,large,haut;
  rectangle(Graphics appG,int x, int y, int l, int h)
  { g=appG;
    orx=x; ory=y; large=l; haut=h;}
  public void dessine()
  { g.setColor(Color.black);
    g.drawRect(orx,ory,large,haut);}
  public void bouge (int x, int y)
  { orx +=x; ory+=y;}
}
//*****
class rectplein extends rectangle
{ Color couleur; //variable supplémentaire
  rectplein (Graphics appG,int x, int y, int l, int h, Color col)
  { super(appG,x,y,l,h); //appel constructeur de la classe rectangle
    couleur=col; } //initialisation de la nouvelle variable
  public void dessine()
  { g.setColor(couleur);
    g.fillRect(orx,ory,large,haut);
    g.setColor(Color.black);
    g.drawRect(orx,ory,large,haut);}
}
//*****
public class testrect extends Applet
{ rectangle r1;
  rectplein rp1;
  public void init()
  { setBackground(Color.lightGray);
```

```

r1= new rectangle(getGraphics(),10,20,100,50);
rp1= new rectplein(getGraphics(),100,100,100,50,Color.red);}
public void paint(Graphics g)
{ r1.dessine();
  rp1.dessine(); //appel méthode de "rectplein"
  rp1.bouge(30,60); //appel méthode de "rectangle"
  rp1.dessine();}
}

```



Arguments des méthodes.

Beaucoup de langages autorisent le passage des arguments soit par **valeur** soit par **adresse**. Dans le cas d'un passage d'argument par valeur, on fait une copie de la valeur des paramètres dans une pile et c'est l'adresse mémoire de cette pile qui est transmise à la méthode. Les adresses réelles des arguments étant inconnues par la méthode, celle-ci ne peut pas les modifier.

Dans le cas d'un passage d'argument par adresse, on transmet à la méthode l'adresse mémoire (pointeur) de la variable argument : les modifications apportées à la valeur de la variable par la méthode sont immédiates et définitives.

Pour JAVA, le mode de passage est fonction de la nature des arguments :

Les arguments de types simples (entiers, réels, caractères ...) sont passés par valeur et les arguments d'objets ou de tableaux sont passés par adresse.

Si l'on souhaite qu'une méthode modifie une variable de type simple, il ne faut pas passer celle-ci comme argument mais il faut la déclarer variable de classe.

Syntaxe pour les arguments de type tableau

En-tête de la méthode : type (ou void) nomDeLaMethode (typeDuTableau nomDuTableauM []) // crochets seuls.

Appel : nomDeLaMethode(nomTableauG) // pas de crochets après le nom du tableau.

Les opérations effectuées par la méthode sur les éléments du tableau "nomDuTableauM" s'appliquent en fait au tableau nomTableauG qui est le seul à avoir une adresse mémoire effective.

4. Structures et modificateurs

4.1 Les packages

4.1.1 Définition

JAVA permet le regroupement de classes dans une bibliothèque de classes appelée « package ». Ils sont utilisés pour les projets importants qui contiennent un nombre de classes important ou pour constituer des bibliothèques de classes réutilisables. Les classes standards du langage sont regroupées dans différents packages.

Pour créer un package, on utilise un fichier source (extension java) contenant l'ensemble du code des différentes classes.

4.1.2 Utilisation des packages

Pour utiliser une classe d'un package, il est possible de faire référence au nom complet de la classe (en utilisant la notation pointée ce qui est très lourd) ou d'importer la classe au moyen du mot clé *import* placé en début de fichier.

Pour importer la classe *Graphics* utilisée pour le dessin, on peut écrire en début de programme :

```
import java.awt.Graphics ;
```

Si on utilise beaucoup de classes d'un package ce système est assez lourd et il est préférable d'importer la totalité du package en utilisant la notation :

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

4.1.2 Les packages standard de JAVA

Toutes les versions de JAVA sont fournies avec une « interface de programmation d'application » (en anglais API acronyme pour : Application Programming Interface) qui regroupe un ensemble de classes d'usage général. Ces classes sont regroupées dans les packages suivants :

java.lang : ce package qui contient les éléments du langage est importée automatiquement.

java.applet : gestion des applets

java.awt : (Abstract Windowing Toolkit) classes pour la mise en place de l'interface utilisateur.

java.awt.image : gestion des images

java.awt.peer : interface avec le système d'exploitation. Réservé aux utilisateurs très très confirmés.

java.util : utilitaires divers.

java.io : gestion des fichiers et des entrées sortie. Ne sera pas étudié ici.

java.net : gestion des accès réseau. Ne sera pas étudié ici.

Sauf java.lang, ces packages doivent être importés par une instruction *import* placée en début de programme.

Le contenu de ces différents packages standard de JAVA sera étudié ultérieurement.

4.2 Les modificateurs

Ce sont des mots clé utilisés lors des déclarations pour modifier les attributs habituels des variables ou des méthodes. Il est possible de les placer avant ou après la déclaration du type.

Seules les variables d'instance et de classe peuvent être qualifiées par un modificateur. Par défaut (pas de modificateur), la visibilité s'étend au package de l'objet de définition.

4.2.1 private, public protected

Ces trois modificateurs peuvent être combinés pour modifier la visibilité (portée) des classes, méthodes, variables. Il existe 4 niveaux différents de visibilité.

public : classes, méthodes, variables déclarées « *public* » sont visibles par toutes les autres méthodes que ce soit à l'intérieur ou à l'extérieur du package de définition. La déclaration de variables publiques est contraire au principe d'encapsulation.

protected : méthodes, variables (*protected* n'est pas autorisé pour les classes) déclarées « *protected* » ne sont accessibles que par les méthodes de la classe et des sous-classe du package de l'objet de définition.

private protected : cette association restreint la visibilité à la classe et à ses sous-classes.

private : c'est le degré de restriction le plus fort ; seules les méthodes de la classe peuvent voir une entité (variable ou méthode) déclarée *private*.

4.2.2 static

Si dans une classe, on déclare une variable « *static* » cette variable va contenir une information commune à toutes les instance de la classe. Une telle variable est dite variable de classe.

Une méthode déclarée statique (*static*) est une méthode qui agit sur les variables de classe.

Exemple :

On peut compléter l'exemple « rectangle » du chapitre 3 en introduisant une notion d'échelle pour le dessin. Le facteur d'échelle qui sera commun à toutes les instances sera déclaré par :

```
private static float echelle = 1.0f
```

Pour mettre à jour cette variable de classe, on peut créer la méthode suivante :

```
static void init_echelle(float ech)
{ echelle = ech;}
```

Cette méthode qui agit sur une variable de classe peut être déclarée *static*.

La méthode "surface" doit être modifiée de la manière suivante :

```
public int surface()
{ int sur = (int)(large*haut*echelle);
  this.dessine();
  g.drawString(""+sur,orx+10,ory+20);
  return sur;}
```

Pour appeler la méthode *init_echelle* la syntaxe est :

```
rectangle.init_echelle(0.75f);
```

En effet comme la méthode *init_echelle* a été déclarée *static* elle ne référence aucune instance de la classe. On préfixe donc le nom de la méthode par le nom de la classe.

Remarque :

Si l'on omet le modificateur *static* dans la déclaration de la méthode *init_echelle*, pour pouvoir invoquer cette méthode il faut la préfixer par le nom d'une instance existante ce qui n'est pas très logique. L'appel doit alors s'écrire :

```
r1.init_echelle(0.75f);
```

mais si l'instance *r2* existe alors « *r2.init_echelle(0.75f);* » peut également être utilisé.

4.2.3 final

Variables : une variable déclarée « *final* » est en fait une constante. Il n'est plus possible de la modifier.

Après la déclaration : `final int x = 20 ;` le code `x = 30 ;` déclenche une erreur de compilation.

Méthodes : les méthodes déclarées « *final* » ne peuvent pas être remplacées dans une sous-classe.

Classes : les classes déclarées « *final* » ne peuvent pas avoir de sous-classe.

4.2.4 abstract

Les classes déclarées abstraites ne peuvent pas créer d'instances. Une classe abstraite est un cadre général utilisé pour générer des sous-classes ayant globalement la même structure. Une classe abstraite peut comporter des méthodes classiques dont hériteront toutes ces filles et des méthodes abstraites qui devront être implémentées de manière spécifique dans les classes filles. Les déclarations « *abstract* » sont donc destinées uniquement à améliorer la structuration du code.

4.2.5 synchronized

Dans les programmes qui gèrent plusieurs processus simultanés, il est nécessaire de protéger l'accès aux données communes aux différents processus sinon il peut se produire des conflits d'accès simultané à la même variable. Le modificateur « *synchronized* » est utilisé pour interdire ces accès simultanés.

4.2.6 volatile

Les données susceptibles d'être modifiées de manière externe (par exemple par un périphérique relié à l'ordinateur hôte) sont déclarées « *volatile* ».

Un concepteur d'applets n'aura sans doute jamais besoin d'utiliser ces trois derniers modificateurs.

5. Les applets

Le code des applets est chargé depuis une machine (le serveur) et exécutées sur une autre machine (le client). Une applet est appelée à partir d'une page HTML. L'exécution du « code-byte » de l'applet est réalisée par la machine virtuelle du navigateur qui a chargé la page HTML. Les navigateurs récents possèdent en fait un compilateur qui génère à partir du « code-byte » un exécutable en code natif de la machine hôte. Dans ces conditions la vitesse d'exécution est celle d'un programme compilé.

Afin d'assurer la sécurité de l'utilisateur les possibilités des applets sont limitées :

- une applet ne peut pas écrire de fichiers sur la machine du client.

- une applet ne peut communiquer qu'avec la machine serveur.

- une applet ne peut pas lancer de programmes sur le client.

- une applet ne peut pas charger de programmes écrits en langage natif sur le client.

Compte tenu de ces restrictions, JAVA est langage sûr : en principe il ne permet pas l'introduction de virus ni d'investigations indiscrètes sur le contenu de la machine client.

5.1 Création d'une applet simple

5.1.1 En-tête d'une applet

Une **applet est une classe** de JAVA déclarée **public** qui étend la classe `java.applet.Applet`.

```
public nom_applet extends java.applet.Applet
```

Comme en général on utilise des méthodes de la classe `java.applet`, il est plus simple d'importer le package correspondant. L'en-tête prend alors la forme simplifiée suivante :

```
import java.applet.*;
```

```
public nom_applet extends Applet
```

5.1.2 Une applet très simple

Pour faire quelque chose de visible par l'utilisateur, une applet doit comporter au minimum la méthode `paint()`. Dans cet exemple, nous allons uniquement afficher un texte.

```
import java.applet.*;
```

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

```
public message0 extends Applet{  
    public void paint(Graphics g){  
        g.drawString("Premier exemple",20,30);  
    }  
}
```

Après compilation de ce fichier source (de nom `message0.java`), on obtient un fichier exécutable ayant comme nom `message0.class`. Il faut incorporer ce fichier à une page HTML avec la balise `<APPLET>`. Celle-ci doit au minimum contenir le nom de la classe et les dimensions de la fenêtre dans laquelle l'applet va s'exécuter.

```

<html>
  <head>
    <title>message0</title>
  </head>
  <body>
    <applet code=message0.class width=200 height=50 ></applet>
  </body>
</html>

```

Premier exemple

L'applet message0.

Les navigateurs définissent une couleur de fond, une couleur d'écriture et une fonte par défaut. Ainsi dans IE la couleur du fond est un gris pâle alors que dans Netscape 4.x c'est le blanc.

Pour obtenir des résultats reproductibles, il faut toujours commencer par définir ces trois paramètres. On va déclarer une variable de type "Font", une variable taille et entrer le message à afficher dans une chaîne. Lors du chargement d'une applet la méthode **init** () est toujours appelée. Nous allons la **surcharger** en initialisant ces variables. *mafonte* étant un objet Font doit être initialisée par une instruction new; TimesRoman est le nom d'une fonte standard de JAVA, font.BOLD est une constante de JAVA (en fait déclarée : public final static int BOLD) ; taille définit la taille (en points) de la fonte utilisée.

```

import java.applet.*;
import java.awt.*;

public message1 extends Applet
{
  Font mafonte;
  String s = "Second exemple";
  int taille;

  public void init()
  {
    setBackground (Color.lightGray);
    taille = 16;
    mafonte = new Font("TimesRoman",font.BOLD,taille);}

  public void paint(Graphics g)
  {
    g.setFont(mafonte);
    g.setColor(Color.red);
    g.drawString(s,20,30);}
}

```

Second exemple

L'applet message1.

5.2 La balise <APPLET>

5.2.1 Passage de paramètres

Si l'on désire modifier le message à afficher dans la page HTML, il faut modifier le fichier source puis le recompiler et placer le nouveau fichier message1.class sur le serveur. Il est plus simple de modifier le code de notre programme pour qu'il puisse recevoir des informations depuis la page HTML de chargement. Les données sont transmises au moyen des balises

<param >. Dans ces balises, on trouve le nom du paramètre et sa valeur sous la forme d'une chaîne de caractères.

```
<html>
  <head>
    <title>message0</title>
  </head>
  <body>
    <applet code = message0.class width = 200 height = 50 >
      <param name = "Mes" value="Message">
      <param name = "Tail" value="14">
    </applet>
  </body>
</html>
```

Dans le programme JAVA, avec la méthode `getParameter(nom_du_parametre)` on récupère la chaîne de caractères contenue dans "value".

Mes étant une chaîne, il n'y a pas de conversion de type à réaliser pour la valeur qui est récupérée. Par contre pour la taille de la fonte, il faut convertir une chaîne de caractères en entier. Cette opération est réalisée par la méthode `parseInt` de la classe `Integer` (Classe de `java.lang`). Les conversions de type seront examinées au chapitre 6.

```
import java.applet.*;
import java.awt.*;

public message2 extends Applet
{
  Font mafonte;
  String s ;
  int taille;

  public void init()
  {
    setBackground (Color.lightGray);
    s = getParameter("mes"); //noms insensibles à la casse
    taille = Integer.parseInt(getParameter("Tail"));
    mafonte = new Font("TimesRoman",font.BOLD,taille);}

  public void paint(Graphics g)
  {
    g.setFont(mafonte);
    g.setColor(Color.red);
    g.drawString(s,20,30);} }
```

Nouveau message

L'applet message2 avec "Mes" = "Nouveau message" et "Tail" = "16"

Message nouveau

L'applet message2 avec "Mes" = "Message nouveau" et "Tail" = "20"

5.2.2 Positionnement de l'applet dans la page HTML

L'applet se comporte, pour son positionnement par rapport au texte qui l'entoure, comme une image :

Le paramètre *align* permet de préciser la position de la fenêtre de l'applet dans la page.

De même les paramètres *hspace* et *vspace* définissent un cadre vide autour de l'applet.

La balise <applet> des deux exemples ci-dessus est la suivante :

```
<applet code="message2.class" width="200" height="50" hspace="20"
align="middle"></applet>
```

5.3 Cycle d'une applet

Toute applet hérite de l'objet Applet un certain nombre de méthodes qui sont appelées par le navigateur de manière bien précise lors de son exécution .

init () : cette méthode est exécutée juste après le chargement du code ou un rechargement par le navigateur.

start () cette méthode est invoquée immédiatement après la fin de l'exécution de la méthode init. Cette méthode est aussi invoquée quand la page HTML contenant l'applet redevient visible pour l'utilisateur.

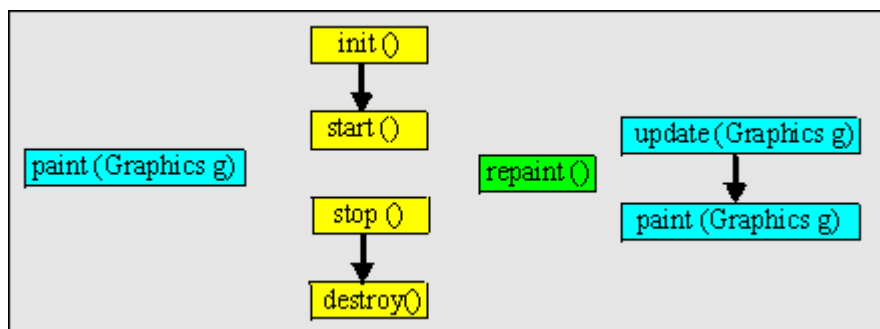
stop () cette méthode est invoquée quand la page HTML disparaît.

destroy () cette méthode est invoquée juste avant la disparition de l'applet afin de pouvoir éventuellement libérer des ressources.

L'applet hérite également de la classe *java.awt.Component* et de la méthode **paint** (Graphics g) qui permet de dessiner dans le composant graphique g attribué à l'applet. Cette méthode est exécutée à la fin de l'exécution de la méthode start.

Si l'applet était masquée (totalement ou partiellement) par une autre fenêtre, elle se redessine lorsqu'elle redevient totalement visible. Ceci est effectué par la méthode **repaint ()**. C'est une méthode qui commence par invoquer la méthode **update** (Graphics g) puis la méthode **paint** (Graphics g). update est une méthode qui peint le composant graphique avec la couleur du fond afin de réaliser son effacement.

La méthode repaint () peut également être invoquée par d'autres méthodes qui vont répondre aux actions de l'utilisateur (souris, clavier ...).



5.4 La classe java.applet

Cette classe possède un certain nombre de méthodes, appelées par le navigateur, qui ne font rien et qui **doivent être surchargées avant de pouvoir être utilisées** : ce sont les méthodes **init ()** **start ()**, **stop ()** et **destroy ()**.

Cette classe possède aussi un certain nombre de méthodes qui utilisent des **URL** (Uniform

Resource **L**ocator) permettant de localiser le serveur de l'applet, des fichiers images et des fichiers sons.

getParameter() : Cette méthode a déjà été étudiée plus avant (§ 5.2.1).

getAppletInfo() : Cette méthode retourne une chaîne de caractères. Elle doit être surchargée comme dans l'exemple sinon elle retourne "null".

getParameterInfo() : Cette méthode retourne un tableau de chaînes de caractères. Elle doit être surchargée comme dans l'exemple sinon elle retourne "null".

Pour chaque paramètre, on peut construire trois chaînes avec par exemple le nom, le type et un commentaire.

getCodeBase() : Cette méthode retourne un URL qui pointe sur le répertoire du serveur qui contient l'applet. Si on désire examiner sa valeur, il faut convertir l'URL en chaîne de caractères avec la méthode `getCodeBase().toString()`.

getDocumentBase() : Cette méthode retourne un URL qui pointe sur le répertoire du serveur qui contient la page HTML de chargement de l'applet. Si on désire examiner sa valeur, il faut convertir l'URL en chaîne de caractères avec `getDocumentBase().toString()`.

getAudioClip() : Cette méthode retourne un **objet AudioClip** qui est en fait un fichier son au format "AU". Comme arguments, il faut indiquer le chemin absolu de fichier (URL) et son nom. Quand le fichier est chargé, il est possible de l'exécuter avec la méthode **play()**.

getImage() : Cette méthode retourne un **objet Image** qui est en fait un fichier au format "GIF" ou "JPG". Comme arguments, il faut indiquer le chemin absolu du fichier (URL) et son nom. Quand le fichier correspondant est chargé, il est possible de visualiser l'image avec la méthode `drawImage()`.

showStatus() : Cette méthode affiche sa chaîne argument dans la barre d'état du navigateur.

L'applet ci-dessous utilise un certain nombre de ces méthodes. En voici le code :

```
import java.applet.*;
import java.awt.*;

public class applet1 extends Applet
{   Font mafonte;           //choix de la fonte
    Image monimage;        //objet image
    String sima="sign141.gif"; //le nom d'un fichier image
    AudioClip bip;         //objet sonore
    String base;
    String [][] paramInfo={{ "par1","1-10","valeur 1"},
                           {"par2","1-5","valeur 2"} }; //declaration et assignation
    String st[][];

    public String getAppletInfo()
    {   return "Auteur: J.J. Rousseau 2001"; }

    public String[][] getParameterInfo()
    {   return paramInfo; }

    public void init()
    {   setBackground (Color.lightGray); //couleur du fond
        mafonte = new Font("Helvetica",0,12);
        st = new String[3][2];
        monimage = getImage(getDocumentBase(),sima);
        base=getCodeBase().toString(); //conversion de l'URL en chaîne
        bip=getAudioClip(getCodeBase(),"ding.au"); //nom du fichier son

    public void paint(Graphics g)
    {   g.setFont(mafonte);
        g.drawString(getAppletInfo(),10,15);
        st=getParameterInfo();
        g.drawString(base,10,30); //adresse du fichier sur le serveur
        g.drawImage(image,20,50,this); /*affichage de l'image "sign141.gif"
            origine en x=20, y=50*/
        g.drawString(st[0][0],150,60); //affiche "par1"
        g.drawString(st[0][1],150,80); //affiche "1-10"
        g.drawString(st[0][2],150,100); //affiche "valeur 1"
        bip.play(); /*execute le fichier "ding.au".
            il faut disposer d'une carte son pour l'entendre.*/
        showStatus("Dans la barre d'état"); //examiner la barre d'état du navigateur.
    }
}
```



Applet correspondant au code ci-dessus.

Placez le pointeur de la souris dans le cadre de l'applet, cliquez et regardez la barre d'état.

6. Le package java.lang

Ces classes ne nécessitent pas d'instruction *import*.

6.1 La classe Math

Cette classe contient des méthodes de calcul sur les entiers et les réels conformes à la norme IEEE 754 ce qui assure leur portabilité sur les clients. Elle contient également des approximations (en double) de e et de π .

Toutes les méthodes doivent être préfixées par le nom de la classe.

Les fonction **abs(x)**, **min(x,y)** et **max(x,y)** admettent les types **int**, **long**, **float** et **double** comme arguments.

```
int i,j = -4;  
double a = 4.2, b = Math.PI, c;  
i = Math.abs(j); //préfixer par Math !!!  
c = Math.min(a,b);
```

Les fonctions trigonométriques, exponentielle, logarithme et puissance utilisent uniquement des **doubles** comme arguments. Si on utilise des arguments illégaux, les méthodes de la classe Math génèrent une exception. Elles retournent NaN (Not a Number).

6.1.1 Fonctions trigonométriques

Ces fonctions travaillent toujours sur des valeurs en radians. On trouve les fonctions :

sin (double) : sinus de l'argument exprimé en radian

cos (double) : cosinus

tan (double) : tangente

asin(double) : arc sinus (valeur dans l'intervalle $[-\pi/2, +\pi/2]$)

acos(double) : arc cos (valeur dans l'intervalle $[0, +\pi]$)

atan (double) : arc tangente (valeur dans l'intervalle $[-\pi/2, +\pi/2]$)

atan2(double) : cette dernière fonction retourne la valeur (en radians) de l'angle θ des coordonnées polaires (r et θ) du point correspondant aux coordonnées cartésiennes qui sont passées comme arguments.

```
double theta = Math.atan2(10,10); // theta = 0.785398... (45°)
```

6.1.2 Autres fonctions

exp (double) : exponentielle

log (double) : logarithme népérien

sqrt (double) : racine carrée

pow (double, double) : élévation du premier argument à la puissance dont la valeur est le second argument. Si le premier argument est nul, le second doit être strictement positif. Si le premier argument est strictement négatif, le second doit être une valeur entière.

random (génération à chaque appel d'un nombre pseudo-aléatoire compris entre 0 et 1.0).

(Il existe également dans le package java.util une classe **Random** qui permet une gestion plus complète des nombres aléatoires.)

On dispose aussi de trois méthodes pour les arrondis : **ceil** (arrondi à l'entier supérieur), **floor** (arrondi à l'entier inférieur), **round** (arrondi par rapport à 0,5). Ces trois méthodes admettent des "double" comme arguments et elles retournent un double sans partie décimale et non pas un entier. **round** admet aussi des arguments de type "float" auquel cas la valeur retournée est du type "int".

```
double x = 0.751, z;  
z = Math.ceil ( x ); // z = 1.0  
z = Math.floor ( x ); // z = 0.0  
z = Math.round ( x ); // z = 1.0
```

6.2 Les classes "enveloppes"

Selon les auteurs ces classes sont aussi nommées types composites ou wrappers.

En plus des types simples **boolean, byte, char, double, float, short, int, long**, il existe en JAVA des classes enveloppes **Boolean, Character, Double, Float, Integer, Long, Number** (une classe abstraite) **et String**. qui concernent des **objets**. **Ces classes ont principalement été développées pour effectuer des conversions de types**. Alors que les types simples sont manipulables avec les opérateurs du langage, les objets des classes enveloppes sont accessibles par des méthodes et nécessitent l'usage d'un constructeur.

Beaucoup de ces méthodes sont déclarées **static** (méthodes de classe) et leurs noms doivent de préférence être **préfixés par le nom de la classe**. Les quelques méthodes d'instance doivent être **préfixées par le nom de la variable d'instance**.

6.2.1 Classe Integer

Voici la description des méthodes les plus utilisées de cette classe :

Constructeur : On définit un objet Integer à partir d'un entier (int) ou d'un objet String.

```
int i = 10;
```

```
String s = "10";
```

```
Integer I = new Integer(i);
```

```
Integer J = new Integer(s);
```

doubleValue() : méthode d'instance qui permet de convertir un objet Integer en double.

floatValue() : méthode d'instance qui permet de convertir un objet Integer en float.

intValue() : méthode d'instance qui permet de convertir un objet Integer en int.

parseInt() : méthode de classe qui convertit une **chaîne en entier de type int**.

valueOf() : méthode de classe qui convertit le premier argument (chaîne) en un objet Integer dans la base de numération donnée par le second argument. Si la base est omise la conversion est faite en base 10.

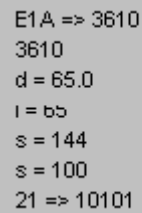
toString() : méthode de classe qui convertit le premier argument (objet Integer) en une chaîne dans la base de numération donnée par le second argument. Si la base est omise la conversion est faite en base 10.

toBinaryString () : méthode de classe qui convertit un objet Integer en une chaîne dans la base 2. [analogue à toString(I , 2)]

On trouvera dans le listing suivant des exemples d'application de ces méthodes.

```
import java.applet.*;
import java.awt.*;
public class wrapper extends Applet
{ String s,sd;
  Integer I;
  int i,j;
  double d;
public void init()
{ setBackground(Color.lightGray);}

public void paint(Graphics g)
{ s="E1A"; sd="65";
  I=Integer.valueOf(s,16);// en Integer (méthode de classe)
  i=I.intValue(); // entier (méthode d'instance)
  g.drawString(s+" => "+i, 10, 20);
  i=Integer.valueOf(s,16).intValue();//en une seule fois
  g.drawString(""+i, 10, 35);
  d=Integer.valueOf(sd).doubleValue();//en double
  g.drawString("d = "+d, 10, 50);
  i=Integer.parseInt(sd); //chaîne en entier
  g.drawString("i = "+i, 10, 65);
  s=Integer.toString(100,8); //entier en chaîne,base 8.
  g.drawString("s = "+s, 10, 80);
  s=Integer.toString(100); //entier en chaîne,base 10.
  g.drawString("s = "+s, 10, 95);
  s=Integer.toBinaryString(21);
  g.drawString("21 => "+s, 10, 110);}
}
```



```
E1A => 3610
3610
d = 65.0
i = 65
s = 144
s = 100
21 => 10101
```

La classe **Long** possède des méthodes analogues. Consulter la documentation de votre version JAVA.

6.2.2 Classe Double

Constructeur : on définit un objet Double à partir d'un réel (double) ou d'une chaîne.

intValue() , **floatValue()** , **longValue()** et **doubleValue()** () sont des méthodes d'instance qui permettent de convertir un objet Double en **int**, **float**, **long** ou **double**.

valueOf() () : méthode de classe qui convertit une chaîne en un objet Double.

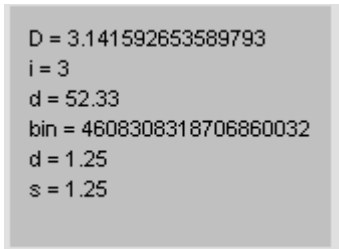
Cette méthode peut être combinée avec la méthode `doubleValue()` pour **convertir une chaîne en double**.

`d = Double.valueOf(s).doubleValue()` ; la partie soulignée assure la conversion de la chaîne en Double, la seconde partie de l'instruction assure la conversion du Double en double.

toString() () : méthode de classe qui convertit un objet Double en une chaîne.

doubleToLongBits() () : méthode de classe qui convertit un double en sa représentation binaire au format IEEE. **longBitsToDouble()** assure la conversion inverse.

On trouvera dans le listing suivant des exemples d'application de ces méthodes.

<pre>import java.applet.*; import java.awt.*; public class wrapperD extends Applet { String s; double d; int i; public void init() { setBackground(Color.lightGray);} public void paint(Graphics g) { d=Math.PI; Double D = new Double(d); g.drawString("D = "+d, 10, 20); i=D.intValue(); // conversion en entier g.drawString("i = "+i, 10, 35); s="52.23"; d=Double.valueOf(s).doubleValue();//chaîne en double g.drawString("d = "+d, 10, 50); d = 1.25; long bin = Double.doubleToLongBits(d); g.drawString("bin = "+bin, 10, 65);//représentation IEEE d = Double.longBitsToDouble(bin); g.drawString("d = "+d, 10, 80);//conversion inverse s=Double.toString(d); //conversion double vers chaîne g.drawString("s = "+s, 10, 95);} }</pre>	
---	--

La classe **Float** possède des méthodes analogues. Consulter la documentation de votre version JAVA.

6.2.3 Classe Character

Constructeur : Le constructeur convertit un char en Character..

charValue () : méthode d'instance qui convertit un Character en char.

toString () : méthode de classe qui convertit un objet Character en chaîne.

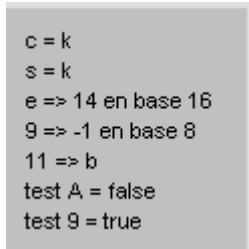
digit () : méthode de classe qui retourne la valeur numérique du Character passé comme premier argument en utilisant le second argument comme base de numération [bases de 2 (MIN_RADIX) à 36 (MAX_RADIX)]. Si la conversion est impossible, la méthode retourne - 1.

forDigit () : méthode de classe qui retourne le char correspondant à l'entier passé comme premier argument en utilisant le second argument comme base de numération (bases de 2 à 36). Si la conversion est impossible, la méthode retourne le char null.

isDigit (), **isLowercase ()**, **isUpperCase ()**, **isSpace ()** sont des méthodes de classes booléennes qui testent si le char passé en argument est un chiffre, une minuscule, une majuscule ou un espace.

toLowerCase () et **toUpperCase ()** : méthodes de classe qui modifient si nécessaire la casse du char passé en argument.

On trouvera dans le listing suivant des exemples d'application de ces méthodes.

<pre>import java.applet.*; import java.awt.*; public class wrapperC extends Applet { char c = 'k'; Character C = new Character(c); String s; public void init() { setBackground(Color.lightGray);} public void paint(Graphics g) { c=C.charValue(); g.drawString("c = "+c, 10, 20); s=C.toString(); g.drawString("s = "+s, 10, 35); int i =Character.digit('e',16); g.drawString("e => "+i+" en base 16", 10, 50); i =Character.digit('e',8); g.drawString("e => "+i+" en base 8", 10, 65); c=Character.forDigit(11,16); g.drawString("11 => "+c, 10, 80); boolean test=Character.isDigit('A'); g.drawString("test A "+test, 10, 95); boolean test=Character.isDigit('9'); g.drawString("test 9 "+test, 10, 110);} }</pre>	 <pre>c = k s = k e => 14 en base 16 9 => -1 en base 8 11 => b test A = false test 9 = true</pre>
---	--

6.3 La classe String

Une chaîne est une suite de caractères placés entre deux guillemets (").

Les chaînes **String** ont une longueur fixe : toute nouvelle affectation d'une chaîne existante entraîne la création par le compilateur d'une nouvelle chaîne dans un emplacement mémoire différent. Si les chaînes String sont faciles à utiliser, elles ne génèrent pas un code efficace. La classe StringBuffer plus lourde à mettre en oeuvre crée un code bien plus efficace que les String.

Contrairement aux autres objets JAVA, il n'est pas nécessaire d'utiliser **new** pour construire une chaîne String.

Les codes `String s = "ABCDE";` et `String s = new String("ABCDE");` sont équivalents.

Examinons quelques méthodes utiles de cette classe :

length () : méthode d'instance qui retourne la longueur de la chaîne.

charAt () : méthode d'instance qui retourne le char qui occupe dans la chaîne s la position précisée par l'argument. Le premier caractère occupe la position 0 et le dernier la position s.length() -1.

concat () : cette méthode d'instance concatène la chaîne passée comme argument à la chaîne utilisée comme préfixe. Pour concaténer deux chaînes il est plus simple d'utiliser le signe +.

equals () : méthode booléenne d'instance qui compare la chaîne passée comme argument à la chaîne utilisée comme préfixe.

Attention on ne peut pas comparer l'égalité de deux chaînes avec " = = ". Pour comparer s1 et s2 il faut utiliser `if (s1.equals(s2))` et non pas `if (s1 == s2)`

indexOf () : méthode d'instance qui retourne la position de la première occurrence du char passée comme argument dans la chaîne utilisée comme préfixe. La méthode retourne -1 si le caractère n'existe pas. La même méthode existe avec deux arguments, le second indique où commencer la recherche. Les mêmes méthodes existent également avec comme premier argument une sous-chaîne de caractères. Les quatre formes de cette méthode existent également pour la méthode analogue **lastIndexOf ()** qui retourne la position de la dernière occurrence de la partie cherchée.

substring () : cette méthode d'instance permet d'extraire une sous-chaîne de la chaîne utilisée comme préfixe. Dans la première syntaxe, la recherche commence à partir de la valeur passée comme argument. Dans la seconde, la recherche s'effectue entre les deux valeurs passées en arguments.

toLowerCase () et toUpperCase () : méthodes d'instance qui modifient si nécessaire la casse de la chaîne utilisée comme préfixe.

trim () : méthode d'instance supprime les caractères "espace" placés en début et en fin de chaîne.

valueOf () : méthodes de classe qui assurent la **conversion de l'argument (booléen, char, int, double, float, double) en chaîne**.

Double.toString(), Integer.toString() ... exécutent la même opération.

Remarque : Dans la méthode drawString () la conversion d'un littéral x est réalisée automatiquement en faisant suivre la chaîne initiale (qui peut être réduite à "") de + x. Cette facilité a déjà été utilisée dans toutes les applets données en exemple.

On trouvera dans le listing suivant des exemples d'application de ces méthodes.

```
import java.applet.*;
import java.awt.*;

public class chaines extends Applet
{ char c;    int i;
  String s1="chAine1",s2,sct;
  Font font;

  public void init()
  { setBackground(Color.lightGray);
    font = new Font("Helvetica",0,11);}

  public void paint(Graphics g)
  { g.setFont(font);
    s2 = "Exemple de chaine";
    g.drawString("s = "+s2, 10, 20);
    i=s2.length();
    g.drawString("longueur = "+i, 10, 35);
    c=s2.charAt(4);// 5 ieme caractère
    g.drawString("char[4] = "+c, 10, 50);
    sct=s1.concat("_chaine2");
    g.drawString("sct = "+sct, 10, 65);
    boolean test = s1.equals(s2);//égalité des chaînes ?
    g.drawString("s1 = s2 : "+test, 10, 80);
    i = s1.indexOf('A');
    g.drawString("position du A = "+i, 10, 95);
    i = s2.indexOf('e',4);
    g.drawString("position du e = "+i, 10, 110);
    sct = s2.substring(5);//syntaxe 1
    g.drawString("s2 (5 à fin) = "+sct, 10, 125);
    sct = s2.substring(4,12);//syntaxe 2
    g.drawString("s2 (4 à 12) = "+sct, 10, 140);
    sct = s1.toUpperCase();//en majuscule
    g.drawString("s1 => "+sct, 10, 155);
    float r = 56.123f;
    sct = String.valueOf(r);
    g.drawString("r = "+sct, 10, 170);
    g.drawString("r = "+r, 10, 185);}//directement
}
```

```
s = Exemple de chaine
longueur = 17
char[4] = p
sct = chAine1_chaine2
s1 = s2 : false
position du A = 2
position du e = 6
s2 (5 à fin) = le de chaine
s2 (4 à 12) = ple de c
s1 => CHAINE1
r = 56.123
r = 56.123
```

6.4 La classe StringBuffer

Lors de la création d'un objet StringBuffer, le compilateur réserve une zone mémoire supérieure à la longueur de la chaîne initiale : il est possible de modifier la chaîne sans procéder à une nouvelle allocation mémoire. On peut considérer qu'un objet StringBuffer est un tableau de caractères accessibles séparément par leurs indices [méthodes charAt() et setCharAt()].

Remarques : Lors de l'usage du signe + pour réaliser la concaténation de chaînes de type String, le compilateur génère un code utilisant la méthode append() de la classe StringBuffer. Les méthodes append() et insert() admettent comme arguments tous les littéraux, des chaînes et des objets.

On trouvera dans le listing suivant des exemples d'application des méthodes de cette classe.

```
import java.applet.*;
import java.awt.*;

public class strbuf extends Applet
{ char c; int i;
  String s1,s2;
  StringBuffer sb;
  Font font;
public void init()
{ setBackground(Color.lightGray);
  font = new Font("Helvetica",0,11);}

public void paint(Graphics g)
{ g.setFont(font);
  sb = new StringBuffer("Exemple de "); //affectation
  g.drawString("sb1 = "+sb, 10, 20);
  sb.append("chaîne"); //concaténation
  g.drawString("sb2 = "+sb, 10, 35);
  c=sb.charAt(4);
  g.drawString("sb[4] = "+c, 10, 50);
  sb.insert(10,"xxx"); //argument chaîne
  g.drawString("sb3 = "+sb, 10, 65);
  sb.setCharAt(10,'%');
  g.drawString("sb4 = "+sb, 10, 80);
  s1="laval";
  g.drawString("s1 = "+s1, 10, 95);
  sb=new StringBuffer(s1);
  s2=sb.reverse().toString(); //conversion en chaîne
  g.drawString("s2 = "+s1, 10, 110);
  boolean test=s1.equals(s2); //test d'égalité des chaînes
  g.drawString(s1+" palindrome ? "+test, 10, 125);
  i=sb.length();
  g.drawString("longueur = "+i, 10, 140);}
}
```

```
sb1 = Exemple de
sb2 = Exemple de chaîne
sb[4] = p
sb3 = Exemple dexxx chaîne
sb4 = Exemple de%xx chaîne
s1 = laval
s2 = laval
laval palindrome ? true
longueur = 5
```

7. Ecrire et dessiner dans les applets

7.1 Ecrire

A partir d'une applet, il est possible d'afficher du texte dans la fenêtre de l'applet ou dans la console JAVA. Beaucoup d'utilisateurs des navigateurs ignorent l'existence de cette fenêtre qui est utilisée par la machine virtuelle pour envoyer ses messages aux utilisateurs. Pour l'activer dans IE, il faut aller dans le menu "Affichage" et dans Netscape, il faut aller dans "Communicator", "Outils". Pour afficher la chaîne de caractères `str` dans la console, la syntaxe est : `System.out.print(str)` (pas de saut de ligne après) ou `System.out.println(str)`.

Il est possible d'utiliser la console pour le débogage des programmes.

Pour afficher du texte dans la fenêtre d'une applet, il suffit d'utiliser la méthode `drawString()` de la classe `Graphics` qui fait parti du package `java.awt`. Cette méthode dessine la chaîne passée comme premier argument à partir du point de coordonnées `(x, y)`. Le préfixe utilisé lors de l'appel à la méthode doit être le nom du contexte graphique (nom de l'instance de la classe `Graphics` en cours d'utilisation).

Les caractères de contrôle comme `'\r'` (CR), `'\n'` (NL), ... sont fonctionnels dans la console JAVA mais ne sont pas utilisables avec la méthode `drawString()` : pour afficher du texte sur plusieurs lignes, il faut découper la chaîne et préciser les coordonnées de départ de chaque sous-chaîne.

Sauf indications contraires, la couleur d'écriture est la couleur par défaut (en général le noir) et la fonte utilisée est la fonte par défaut (variable selon le navigateur utilisé). Il est toujours préférable de préciser au début de la méthode `init()` la fonte qui sera utilisée.

7.1.1 La classe **Font**

Avant de pouvoir utiliser une fonte, il faut créer une instance d'un objet `Font` et l'initialiser avec les paramètres suivants :

Le nom de la fonte : Les machines virtuelles des navigateurs possèdent en général les fontes "Helvetica" (SansSerif, Arial) "TimesRoman" (Serif) qui sont des fontes proportionnelles et la fonte "Courier" qui possède un espacement fixe.

Le style des caractères : Il existe trois constantes `Font.PLAIN = 0` (normal), `Font.BOLD = 1` (**gras**) et `Font.ITALIC = 2` (*italique*) qui peuvent être combinées.

La taille de la fonte en points.

On trouvera dans la seconde partie de l'exemple ci-dessous comment chercher puis afficher les polices présentes dans la machine virtuelle utilisée. La méthode `getFontList()` de la classe abstraite `Toolkit` retourne un tableau de chaînes de caractères contenant les noms des fontes.

7.1.2 La classe **FontMetrics**

Afin de pouvoir gérer l'affichage, on dispose de la classe `FontMetrics` dont les méthodes retournent des informations sur les caractéristiques de la fonte en cours d'utilisation. Pour pouvoir utiliser ces méthodes, il faut commencer par créer une instance `FontMetrics` au moyen de la méthode `getFontMetrics()` de la classe `Graphics`. Le préfixe utilisé pour appeler cette méthode doit être le nom du contexte graphique en cours d'utilisation. Chaque fois que la fonte d'affichage est changée il faut bien sûr créer une nouvelle instance `FontMetrics`. Les méthodes les plus utiles de cette classe sont :

`getHeight()` qui retourne la hauteur totale (en pixels) de la fonte, `charWidth()` qui retourne la largeur en pixels du caractère passé en argument et enfin de `stringWidth()` qui retourne la

longueur (en pixels) de la chaîne passée en argument. Cette dernière méthode permet de centrer une chaîne ou comme dans l'exemple de réaliser un affichage complexe qui mélange différentes fontes et différentes couleurs.

```
import java.applet.*;
import java.awt.*;
public class fontes extends Applet
{ Font font;
  String s = "Nom : fontes\r\n" + "Auteur : Rousseau\r\n" +
    "Date : 27-12-2001";
  FontMetrics fm;  int w;

  public void init()
  { setBackground(Color.lightGray);
    font = new Font("Helvetica",0,16);}

  public void paint(Graphics g)
  { System.out.println(s); //affichage dans la console
    g.setColor(Color.black); //couleur du pinceau
    g.setFont(font);      fm=g.getFontMetrics();
    s="Consultez ";      g.drawString(s,10,20);
    w=fm.stringWidth(s)+10;
    font = new Font("TimesRoman",1,22);
    g.setFont(font);      fm=g.getFontMetrics();
    g.setColor(Color.red);
    s = "la console ";
    g.drawString(s,w,20);  w+=fm.stringWidth(s);
    font = new Font("Helvetica",2,20);
    g.setColor(Color.black);
    g.setFont(font);      g.drawString("JAVA",w,20);
    g.setColor(Color.blue);
    String polices[]=getToolkit().getFontList();
    font = new Font("TimesRoman",0,12);
    for (int i=0; i<polices.length; i++)
    { g.setFont(font);
      g.drawString(polices[i],10,60+15*i);
      Font ftemp=new Font(polices[i],0,12);
      g.setFont(ftemp);
      g.drawString("Exemple ",150,60+15*i);} }
}
```

Consultez **la console** JAVA

Dialog	Exemple
SansSerif	Exemple
Serif	Exemple
Monospaced	Exemple
Helvetica	Exemple
TimesRoman	Exemple
Courier	Exemple
DialogInput	Exemple
ZapfDingbats	→☒⌘○□●⌘

7.2 La classe Color

La couleur du pinceau dans l'objet graphique g est celle de l'objet Color activé par l'instruction **g.setColor()**.

JAVA utilise la synthèse additive rouge, vert, bleu. Les couleurs sont codées sur 24 bits en utilisant 8 bits pour chaque couleur fondamentale. Le nombre de couleurs possibles est donc $256*256*256 = 16777216$. Le nombre de couleurs effectivement affichées dépend de la carte graphique du client.

Pour définir une couleur, on utilise le constructeur **Color(R, V, B)**. Les arguments sont soit des entiers (compris entre 0 et 255) soit des flottants (compris entre 0.0f et 1.0f).

Color macouleur = new Color (0, 15, 100);

La classe Color possède 12 couleurs prédéfinies :

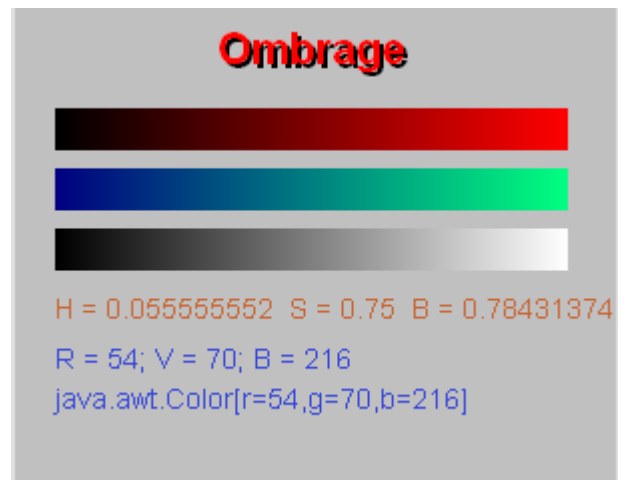
Couleur	Nom	Rouge	Vert	Bleu
Blanc	Color.white	255	255	255
Bleu	Color.blue	0	0	255
Cyan	Color.cyan	0	255	255
Gris pâle	Color.ligthGray	192	192	192
Gris sombre	Color.darkGray	64	64	64
Gris	Color.gray	128	128	128
Magenta	Color.magenta	255	0	255
Noir	Color.black	0	0	0
Orange	Color.orange	255	200	0
Rose	Color.pink	255	175	175
Rouge	Color.red	255	0	0
Vert	Color.green	0	255	0

Les graphistes utilisent en général le système **HSB** : hue, saturation, brightness (teinte, saturation, luminosité). Les méthodes `HSBtoRGB()` et `RGBtoHSB()` permettent d'effectuer les conversions entre les deux systèmes. La méthode `RGBtoHSB` retourne les valeurs HSB dans un tableau de flottants passé en argument (consulter l'exemple pour la syntaxe). Pour plus de détails sur le système HSB [consulter cette page](#).

L'applet suivante donne un exemple d'utilisation des objets FontMetrics , des exemples de création de couleurs en RVB et HSB et d'utilisation des méthodes de cette classe.

```
import java.applet.*;
import java.awt.*;
public class couleurs extends Applet
{ Font font; FontMetrics fm;
  String s,s1 = "Ombrage";
  Color col; int w;
  float hsb[]=new float[3];

  public void init()
  { setBackground(Color.lightGray);
    font = new Font("Helvetica",1,22);}
  public void paint(Graphics g)
  { g.setFont(font); fm=g.getFontMetrics();
    g.setColor(Color.black);
    w=(size().width-fm.stringWidth(s1))/2;
    g.drawString(s1,w,30);
    g.setColor(Color.red);
    g.drawString(s1,w-2,28); //affichages décalés
    font = new Font("Helvetica",0,14); g.setFont(font);
    for (int i=0; i<256; i++) //gamme des rouges
    { col = new Color(i,0,0); g.setColor(col);
      g.drawLine(i+20,50,i+20,70);}
    for (int i=0; i<256; i++)
    { col = new Color(0,i,128); g.setColor(col);
      g.drawLine(i+20,80,i+20,100);}
    for (int i=0; i<256; i++) //gamme des gris
    { col = new Color(i,i,i); g.setColor(col);
      g.drawLine(i+20,110,i+20,130);}
    col = new Color(200,100,50); g.setColor(col);
    Color.RGBtoHSB(200,100,50,hsb);
    s="H = "+hsb[0]+" S = "+hsb[1]+" B = "+hsb[2];
    g.drawString(s,20,155);
    col = new Color(Color.HSBtoRGB(0.65f,0.75f,0.85f));
    int r=col.getRed(); int v=col.getGreen();
    int b=col.getBlue();
    s="R = "+r+" V = "+v+" B = "+b;
    g.setColor(col); g.drawString(s,20,180);
    g.drawString(col.toString(),20,200);}
}
```



7.3 Les méthodes de dessin de la classe Graphics

La feuille de dessin est une matrice de points (pixels). Les coordonnées du coin supérieur gauche sont (0, 0). La méthode size() retourne la taille de la feuille . Les coordonnées du coin inférieur droit sont donc size().width et, size().height.

Attention : La méthode **resize()** ne fonctionne pas avec les applets :

La taille de la feuille est imposée par les valeurs de width et height de la balise <Applet>

Toutes les méthodes de dessin ci-dessous fonctionnent également dans un tampon mémoire. Toutefois j'ai constaté que l'utilisation d'entiers supérieurs à 32768 provoque alors des dépassements de capacité (non signalés par la machine virtuelle !).

Lors de l'utilisation des méthodes de dessin dans un tampon, il semblerait que les entiers **int** sont codés sur seulement 2 octets ???

La méthode

setBackground() permet d'imposer la couleur du fond.

drawLine() : Méthode utilisée pour le tracé de droites. Les 4 paramètres sont les coordonnées des extrémités de la droite.

Il n'existe pas de méthode spécifique **pour allumer un seul pixel** : il faut utiliser la méthode **drawLine** avec des coordonnées de début identiques à celles de fin.

Il n'existe pas dans les différentes versions 1.X de JAVA de méthodes pour modifier le style des traits (épaisseur, tirets, pointillés ...)

drawRect() utilise 4 paramètres : les deux coordonnées du coin supérieur gauche, la largeur et la hauteur du rectangle.

fillRect() remplit le rectangle avec la couleur actuelle du pinceau.

clearRect() utilise la couleur du fond pour peindre le rectangle. Pratique pour effacer une partie de la feuille.

drawRoundRect() et **fillRoundRect()** dessinent des rectangles aux coins arrondis : il faut ajouter deux paramètres supplémentaires pour préciser les dimensions de l'arrondi.

clipRect() est une méthode extrêmement puissante qui supprime les portions des tracés situées à l'extérieur du rectangle défini par la méthode.

Pour utiliser plusieurs zones de clipping, il faut terminer le dessin dans la première zone, **faire un appel à `getGraphics()` pour réinitialiser le contexte graphique** puis définir une nouvelle zone avec **clipRect()** et dessiner dans la nouvelle zone ...

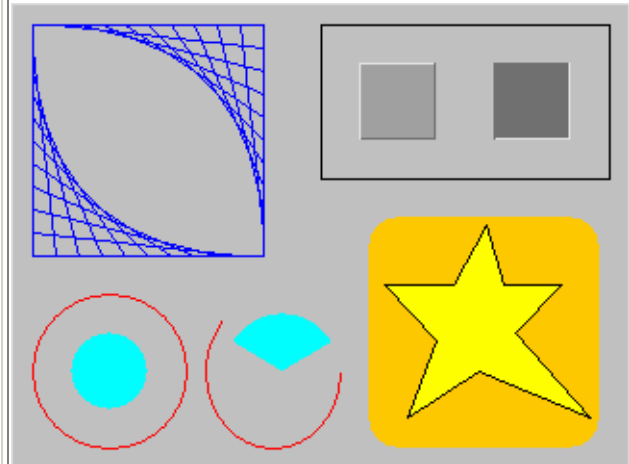
Les cercles, ellipses, arcs peuvent s'inscrire dans un rectangle : leurs méthodes de tracé [**drawOval()**, **fillOval()**, **drawArc()**, **fillArc()**] utilisent les coordonnées de ce rectangle comme paramètres.

Pour le tracé des arcs, il faut préciser en outre l'angle de départ de l'arc en degrés (0 correspond à trois heures, 90 à midi ...) et la longueur de l'arc (toujours en degrés). Si la longueur de l'arc est positive on suit l'arc dans le sens direct.

Les méthodes **drawPolygon()** et **fillPolygon()** utilisent deux tableaux contenant les coordonnées des sommets du polygone et le nombre de sommets à prendre en compte. Le dernier sommet est toujours relié au premier.

Ces méthodes sont illustrées par l'exemple ci-dessous.

```
import java.applet.*;
import java.awt.*;
public class dessins extends Applet
{ Color col = new Color(160,160,160);
  int X[]={ 246,229,193,220,205,242,300,261,285,255 };
  int Y[]={ 114,145,145,174,214,190,214,170,145,145 };
  public void init()
  { setBackground(Color.lightGray);}
  public void paint(Graphics g)
  { g.setColor(Color.blue);
    for (int i=0; i<11; i++)
      g.drawLine(10,10+12*i,10+12*i,130);
    for (int i=0; i<11; i++)
      g.drawLine(130,10+12*i,10+12*i,10);
    g.setColor(Color.black);
    g.drawRect(160,10,150,80);
    g.setColor(col);
    g.fill3DRect(180,30,40,40,true);
    g.fill3DRect(250,30,40,40,false);
    g.setColor(Color.red);
    g.drawOval(10,150,80,80);
    g.drawArc(100,150,70,80,0,-220);
    g.setColor(Color.cyan);
    g.fillOval(30,170,40,40);
    g.fillArc(110,160,60,60,30,120);
    g.setColor(Color.orange);
    g.fillRoundRect(185,110,120,120,30,30);
    g.setColor(Color.yellow);
    g.fillPolygon(X,Y,10);
    g.setColor(Color.black);
    g.drawPolygon(X,Y,10);
  }
}
```



8. Animation des applets

8.1 Les Threads

Avec un langage non multitâche, on réalise une animation en créant une boucle sans fin dans laquelle on met régulièrement à jour l'affichage. Cette méthode présente l'inconvénient majeur d'occuper 100% du temps du processeur.

JAVA peut gérer plusieurs tâches simultanées : Lorsque la machine virtuelle exécute un applet classique elle gère en fait plusieurs processus dont l'exécution du code de l'applet et la gestion dynamique de la mémoire. Pour réaliser une animation, il faut ajouter à l'applet un processus (**Thread**) supplémentaire.

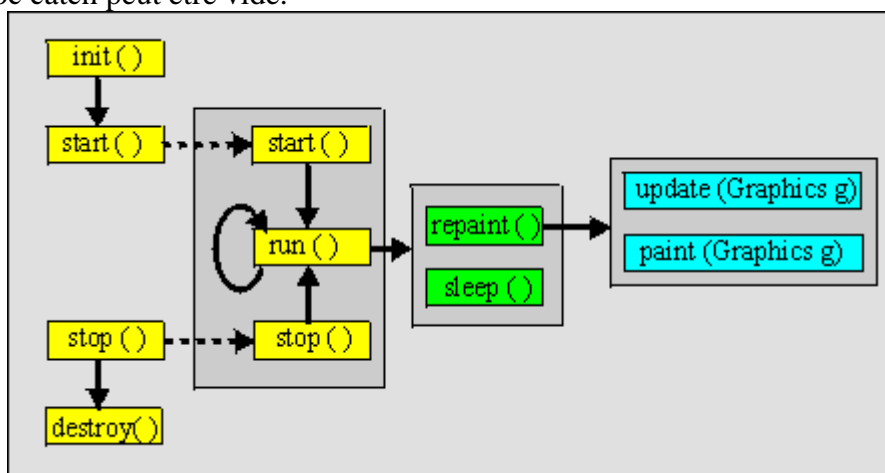
Un thread possède son propre flot d'instruction indépendant de celui de l'applet et dont le déroulement est le suivant. Une méthode **start()** démarre la méthode **run()** qui contient le code à exécuter par le processus. Une méthode **stop()** permet de terminer l'exécution de la méthode **run()**. Pendant son exécution un thread peut être endormi pour une certaine durée [**sleep()**], mis en attente [**suspend()**], réactivé [**resume()**].

Rappel : Le navigateur lance la méthode **start()** à chaque fois que le cadre de l'applet devient visible et la méthode **stop()** quand ce cadre disparaît de l'affichage.

8.2 Modèle d'animation simple

Pour la mise en place d'un thread, on commence par ajouter dans l'entête de l'applet "**implements Runnable**" qui signifie que l'on va implémenter la méthode abstraite `run()` de la classe `Runnable` (une interface de `java.lang`). On crée ensuite comme variable de classe de l'applet une instance d'un objet Thread. On surcharge les méthodes `start()` et `stop()`. Dans la méthode `start()`, on teste l'existence du thread : on le crée s'il n'existe pas puis on active la méthode `start()` par `[nom_thread.start()]` de ce thread [lancement de la méthode `run()`] puis on implémente la méthode `run()`.

Pour un processus très simple, la méthode `run()` peut se résumer à une boucle sans fin qui appelle successivement la méthode `repaint()` de l'applet puis met le thread en sommeil. L'appel à la méthode `sleep` peut générer une exception : il faut impérativement traiter celle-ci au moyen de la séquence `try, catch` prévue pour le traitement des exceptions en JAVA. Par contre le bloc `catch` peut être vide.



Dans l'animation proposée en exemple, on se contente de déplacer une ellipse.

```
import java.applet.*;
import java.awt.*;
public class animal extends Applet implements Runnable
{ Thread runner = null; //variables de classe
  double t;
  public void init()
  { setBackground (Color.lightGray);}
  public void paint(Graphics g)
  { int X=140+(int)(120*Math.sin(t)); //calcul de la position
    t+=0.05; //déplacement lors de l'appel suivant
    g.fillOval(X,20,50,100);}
  public void start() //surcharge de la méthode
  { if (runner == null) //test d'existence
    { runner = new Thread(this); //création , this désigne l'applet
      runner.start();} // lancement de la méthode run()
  }
  public void stop()
  { if (runner != null)
    { runner.stop();
      runner = null;} //destruction
  }
  public void run()
  { while (true) // boucle sans fin
    { try //action à réaliser
      { repaint(); //redessiner
        Thread.sleep(20);} //pause de 20 ms
      catch (InterruptedException e)
      { stop();}} //traitement (facultatif) de l'exception
  }
}
```

J'ai fait volontairement le choix de présenter une animation dont le résultat n'est pas très satisfaisant. L'image scintille et des traces grises traversent régulièrement l'ellipse. Pour comprendre l'origine de ce phénomène, il faut se souvenir que la méthode `repaint()` commence par appeler la méthode `update(Graphics g)` qui efface le cadre de l'applet. Comme cet effacement qui est à l'origine des défauts constatés la solution est de **surcharger la méthode `update()`**.

8.3 La méthode du double tampon

Les auteurs de manuels sur JAVA proposent différentes méthodes pour remédier à ces défauts : effacer uniquement les parties mobiles, clipping, double tampon ... Après avoir expérimenté toutes ces techniques, je considère que seule la méthode du double tampon est efficace à 100 % et que les autres sont du bricolage souvent plus difficile à mettre en oeuvre que cette méthode.

Principe : Le principe de cette technique d'affichage consiste à préparer la nouvelle image à visualiser dans un tampon (zone de la mémoire) puis à l'afficher à l'écran quand elle est entièrement terminée. L'affichage correspond à la copie d'une zone mémoire dans une autre. Tous les processeurs modernes sont dotés d'instructions spécifiques pour effectuer ces transferts et leur durée est très brève.

Méthode à utiliser :

Il faut déclarer une image et le contexte graphique associé. Ces deux objets sont ensuite créés dans `init()` car c'est à ce moment seulement que la taille de la fenêtre de l'applet est connue. Le dessin est réalisé dans le nouveau contexte graphique par la méthode `paint()`. Pour terminer celle-ci, on transfère l'image préparée dans le contexte graphique de l'applet avec la méthode `drawImage()`. Les arguments de cette méthode sont le nom de la zone mémoire, les coordonnées du coin supérieur gauche, la couleur du fond et un objet observateur d'image. Comme ici on a la certitude que la totalité de l'image réside en mémoire, on peut se contenter de celui de l'applet (utilisation de `this`). **Il faut ensuite surcharger la méthode `update()`**.

Si la dimension de la fenêtre de votre navigateur est suffisante, vous pouvez comparer les deux applets.

```
import java.applet.*;
import java.awt.*;
public class anima2 extends Applet implements Runnable
{ private Thread runner = null;
  Image ima;   Graphics h; //déclarations
  int la,ha;
  double t;
  public void init()
  { setBackground (Color.lightGray);
    la=size().width;   ha=size().height; //taille de l'applet
    ima=createImage(la,ha); //creation d'une zone la*ha pixels
    h=ima.getGraphics();} //contexte graphique associé à l'image
  public void update(Graphics g) //surcharge indispensable
  { paint(g);}
  public void paint(Graphics g)
  { h.clearRect(0,0,la,ha); //effacement de la zone de dessin
    int X=140+(int)(120*Math.sin(t));
    t+=0.05;
    h.fillOval(X,20,50,100);
    g.drawImage(ima,0,0,Color.white,this);} //transfert de ima vers l'écran
  //les méthodes start, stop et run sont identiques à celles de l'exemple précédent
  public void destroy() //voir la remarque ci-dessous
  { h.dispose();   ima.flush();}
}
```

Cette technique peut également être utilisée (même en dehors des animations) à chaque fois que l'on oblige l'applet à se redessiner avec une fréquence élevée.

Remarques :

1. Le gestionnaire de mémoire (ramasse-miettes) ne gère pas automatiquement la destruction des objets Images ni des contextes graphiques. L'utilisation de la méthode **destroy()** permet au programmeur de terminer proprement son programme en assurant une libération correcte de la mémoire avec les méthodes **dispose()** et **flush()**.
2. J'ai suivi l'usage de nommer cette technique "double tampon" alors que l'on utilise en fait un seul tampon mémoire.

8.4 Animation avec des images.

Il est aussi possible de réaliser des animations en utilisant l'affichage successif d'images. On obtient un effet analogue à celui des images gif animées. La difficulté de cette méthode provient de ce qu'il faut s'assurer que les images ont bien été chargées à partir du serveur avant de lancer leur affichage sur le client.

Dans le listing ci-dessous on retrouve les techniques examinées plus haut. On pourra noter que les initialisations sont réalisées ici au début de la méthode `run()` et pas dans une méthode `init()`. Le boolean `ok` est mis à "true" quand le chargement de la totalité des images à partir du serveur est terminé. Le contrôle du chargement est effectué par un objet de la classe `MediaTracker` du package `java.awt`. Dans notre exemple, les images sont stockées (sur le serveur) dans le sous-répertoire "image" du répertoire de la page html de chargement de l'applet. Pour avoir un affichage correct, il est essentiel que toutes les images aient les mêmes dimensions. Chaque image recouvrant exactement la précédente, il est en principe inutile d'effacer.

Les méthodes sleep() du Thread et waitForAll du MediaTracker pouvant être à l'origine d'une exception, il faut utiliser des blocs try et catch.

```
import java.applet.*;
import java.awt.*;
public class anima3 extends Applet implements Runnable
{ Thread runner = null;
  Graphics h;  Image imag[];
  int index,large = 0,haut = 0;
  boolean ok = false;
  final int nbima = 10;
  private void displayImage(Graphics g)
  { if (!ok) return;
    g.drawImage(imag[index],(size().width - large)/2,(size().height - haut)/2, this); //centrage de l'affichage
  }
  public void paint(Graphics g)
  { if (ok){ //chargement terminé
    //g.clearRect(0, 0, size().width, size().height); effacement si surcharge de update()
    displayImage(g);}
    else g.drawString("Chargement",10,20);}
  public void start()
  { if (runner == null){
    runner = new Thread(this);
    runner.start();} }
  public void stop()
  { if (runner != null){
    runner.stop();
    runner = null;}}
  public void run()
  { index = 0;
    if (!ok) // chargement des images
    { repaint();
      h = getGraphics();
      imag = new Image[nbima]; //tableau d'images
      MediaTracker tracker = new MediaTracker(this); //
      String strImage;
      for (int i = 0; i < nbima; i++){ //boucle de chargement
        strImage = "image/ima" + i + ".gif"; //nom du fichier
        imag[i] = getImage(getDocumentBase(), strImage); //détermination de son adresse
        tracker.addImage(imag[i], 0); //chargement
      }
      try{
        tracker.waitForAll(); //attente de réalisation de la totalité de la boucle
        ok = !tracker.isErrorAny();}
      catch (InterruptedException e){ }
      large = imag[0].getWidth(this); //dimensions des images
      haut = imag[0].getHeight(this);
      repaint();
      while (true){ //boucle d'animation
        try{
          displayImage(h);
          index++;
          if (index == nbima) index = 0;
          Thread.sleep(100);} //100 ms entre chaque image
        catch (InterruptedException e){stop();}} }
  }
  public void destroy() //pour quitter proprement
  { h.dispose();
    for (int i = 0; i < nbima; i++) imag[i].flush();}
}
```

8.5 Activités indépendantes

Il est possible de gérer de manière simultanée plusieurs processus qui peuvent être indépendants en étendant la classe Thread. Afin de ne pas trop allonger le listing de l'exemple, les deux processus mis en place utilisent la même classe (process) mais les deux activités sont totalement indépendantes et asynchrones. Dans le premier processus, on déplace un rectangle noir de droite à gauche; dans le second on déplace un rectangle rouge de haut en bas. Dans un cas concret, on pourra bien sûr utiliser des classes différentes pouvant agir sur des objets différents.

La méthode run() de l'applet crée deux instances p1 et p2 de la classe process puis lance leur exécution avec p1.start() et p2.start(). La classe process modifie dans sa méthode run() la position de l'origine des rectangles qui lui sont passés en argument. Cette méthode exécute ensuite une boucle infinie qui redessine l'applet. La méthode sleep() d'un Thread pouvant être à l'origine d'une exception, il faut utiliser des blocs try et catch. L'usage d'un double tampon permettrait d'améliorer cette animation.

```

import java.applet.*;
import java.awt.*;
class process extends Thread
{   Rectangle rp; //objet à modifier
    boolean ver;
    process(Rectangle r,boolean v) //constructeur
    {   rp=r;   ver=v;}
    public void run()
    {   int del =(ver) ? 20 : 40; //pour avoir des processus asynchrones
        while(true){
            for (int i=0; i<100; i++){
                if (ver) rp.x=2*i; else rp.y=i; //déplacement dans un sens
                try{Thread.sleep(del);}
                catch (InterruptedException e){ }}
            for (int i=100; i>0; i--){ //puis dans l'autre
                if (ver) rp.x=2*i; else rp.y=i;
                try{Thread.sleep(del);}
                catch (InterruptedException e){ }}}
}
public class anima4 extends Applet implements Runnable
{   Thread runner = null;
    Rectangle r=new Rectangle(0,0,40,20);
    Rectangle r2=new Rectangle(0,0,20,40);
    process p1,p2;
    public void init()
    {   setBackground(Color.lightGray);}
    public void paint(Graphics g)
    {   g.setColor(Color.black);
        g.fillRect(10+r.x,60+r.y,r.width,r.height);
        g.setColor(Color.red);
        g.fillRect(100+r2.x,10+r2.y,r2.width,r2.height);}
    public void start()
    {   if (runner == null)
        {   runner = new Thread(this);
            runner.start();}}
    public void stop()
    {   if (runner != null)
        {   runner.stop();
            runner = null;}}
    public void run()
    {   p1=new process(r,true);
        p2=new process(r2,false);
        p1.start();
        p2.start();
        while (true){
            try{
                repaint();
                Thread.sleep(10);}
            catch (InterruptedException e){ }}}
}

```

9. Gestion du clavier et de la souris

Lorsque l'utilisateur d'une applet appuie sur une touche du clavier, déplace la souris, clique sur un bouton, le système d'exploitation de sa machine déclenche un **événement**. Pour capturer les événements les concepteurs de JAVA ont introduit la classe **Component**. C'est la super-classe abstraite de la plupart des classes de l' "*Abstract Window Toolkit*" qui seront étudiées par la suite. (Les classes de Component représentent des entités ayant une position, une taille, pouvant être dessinées à l'écran ou pouvant recevoir des événements). La classe **Applet** hérite de la classe Component et peut traiter les événements au moyen des méthodes de la classe **Event** de java.awt

A partir de la version 1.1 JAVA n'utilise plus la classe Event pour la gestion du clavier et de la souris !

9.1 Gestion du clavier (Java 1.0.x)

Pour récupérer les événements du clavier, il faut surcharger la méthode **public boolean keyDown(Event evt, int key)**. L'entier associé à la touche enfoncée est retourné dans la variable d'instance key. La classe Event définit des constantes pour les touches non alphanumériques du clavier. Pour tester les touches de modification du clavier, il existe également les méthodes **shiftDown()** (touche [majuscule]), **controlDown()** (touche [Ctrl]) et **metaDown()** (touche [Méta] ou [Alt]). La touche META est utilisée sur les systèmes UNIX. Son équivalent sur les systèmes Apple est la touche [Pomme].

Attention : Si l'applet n'a pas le *focus* elle ne peut pas récupérer les événements du clavier. Pour donner le focus à une applet il faut par exemple cliquer avec la souris dans son cadre ou utiliser la méthode **requestFocus()** comme dans l'exemple ci-dessous. Si vous cliquez en dehors du cadre de l'applet, celle-ci perd le focus.

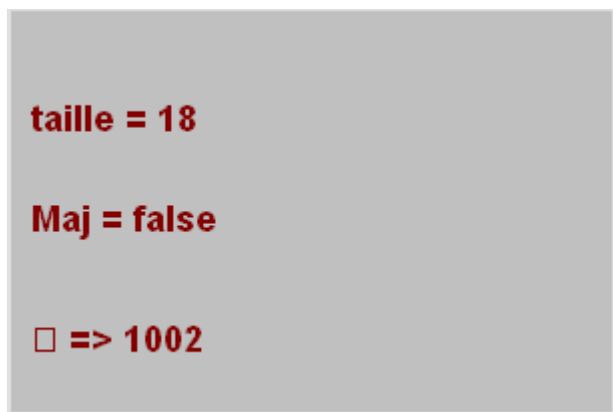
Constantes associées aux touches non alphanumériques : Event.UP = 1004, Event.DOWN = 1005, Event.LEFT = 1006, Event.RIGHT = 1007 (flèches); Event.HOME, Event.END, Event.PGUP, Event.PGDN, Event.F1 à Event.F12.

Constantes associées aux touches modificatrices : ALT_MASK = 8, CTRL_MASK = 2, META_MASK = 4, SHIFT_MASK = 0;

```

import java.applet.*;
import java.awt.*;
public class clavier extends Applet
{ char touche='@';
  int taille=20,code=touche,gras=0,c=125;
  boolean maj;
  public void init()
  { setBackground(Color.lightGray);
    this.requestFocus(); } //donne le focus
  public void paint(Graphics g)
  { Color col = new Color(c,0,0);
    Font font = new Font("Helvetica",gras,taille);
    g.setFont (font);    g.setColor(col);
    g.drawString("taille = "+taille, 10, 60);
    g.drawString("Maj = "+maj, 10, 110);
    g.drawString(String.valueOf(touche)+" => "+code,10,170);}
  public boolean keyDown(Event evt, int key)//surcharge de la méthode
  { switch(key){
    case Event.DOWN : //constante pour la flèche bas
      if (taille>6) taille--; break;
    case 1004 : //constante pour la flèche haut
      if (taille<55) taille++; break;
    case Event.LEFT : if (c>0) c-=5; break;
    case Event.RIGHT : if (c<255) c+=5; break;
    case Event.PGUP : gras=1; break;
    case Event.PGDN : gras=0; break;}
    touche=(char)key; code=key;
    maj = evt.shiftDown(); //touche majuscule enfoncée ?
    repaint(); //pour voir les modifications
    return true;}
  }
}

```



Utiliser les flèches du clavier pour modifier la taille des caractères et leur couleur

Utiliser les touches PGUP et PGDN pour changer la graisse de la fonte.

Cliquer dans le cadre de l'applet si elle ne répond pas aux événements clavier.

9.2 Utilisation de la souris

Comme pour la gestion du clavier, la gestion de la souris impose la surcharge des méthodes de la classe `Component` dédiées à la souris. Ces six méthodes sont les suivantes :

public boolean mouseMove(Event evt, int x, int y) invoquée quand la souris se déplace sans bouton enfoncé.

public boolean mouseDown(Event evt, int x, int y) invoquée quand un bouton de la souris est enfoncé dans la surface de l'applet.

public boolean mouseUp(Event evt, int x, int y) invoquée quand un bouton de la souris est relâché dans la surface de l'applet.

public boolean mouseDrag(Event evt, int x, int y) invoquée quand la souris se déplace avec un bouton enfoncé.

public boolean mouseEnter(Event evt, int x, int y) invoquée quand la souris entre dans la surface de l'applet.

public boolean mouseExit(Event evt, int x, int y) invoquée quand la souris sort dans la surface de l'applet.

Ces 6 méthodes possèdent les mêmes arguments : une instance de la classe **Event**, et les coordonnées (en pixels) du pointeur. Il est inutile de surcharger les méthodes qui ne sont pas utilisées.

Pour rendre JAVA indépendant de la plate-forme utilisée, seule la souris à un bouton est supportée. Toutefois afin de pouvoir utiliser (ou simuler) une souris à deux boutons, la variable d'instance *modifiers* de l'objet `Event` (nommé par exemple `evt`) peut contenir une valeur de modificateur clavier. Pour les systèmes qui gèrent normalement une souris à deux boutons, il est possible de tester si le bouton droit est enfoncé avec `evt.modifiers == Event.META_MASK`.

De même, il est possible de tester si une touche de modification est enfoncée en même temps que le bouton avec `evt.modifiers == Event.XXXX_MASK`. XXXX correspond à CTRL pour la touche contrôle, SHIFT pour la touche majuscule et META pour la touche Méta .

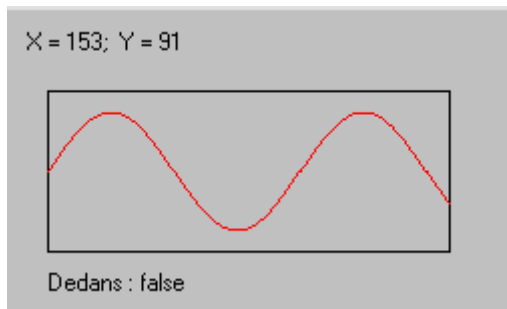
L'exemple ci-dessous utilise toutes les méthodes de gestion de la souris. Il trace en particulier un réticule qui suit les mouvements du pointeur quand celui-ci est à l'intérieur d'un rectangle. On notera l'utilisation de la méthode **r.inside()** de la classe **rectangle** qui permet de déterminer simplement si le pointeur de la souris est à l'intérieur d'une zone rectangulaire donnée.

Il est possible de placer le tracé du réticule dans les méthodes *mouseDown()* et *mouseDrag()* mais il faut alors récupérer le contexte graphique avec l'instruction `Graphics g = getGraphics()` ; il est plus simple d'effectuer tous les tracés à partir de la méthode `paint()` en fonction de la valeur d'un booléen.

```

import java.applet.*;
import java.awt.*;
public class souris0 extends Applet
{
    Font font;
    Rectangle r = new Rectangle(20,40,200,80);
    int X,Y; //coordonnées du pointeur
    int z = r.y + 40; //r.y = coordonnée y du coin sup. gauche de r
    boolean actif,droit,dedans,ctrl;
    public void init()
    {
        setBackground(Color.lightGray);
        font = new Font("Helvetica",0,11);}
    public void paint(Graphics g)
    {
        double t=0;
        g.setColor(Color.black);
        g.drawString("Dedans "+dedans,20,150);
        g.drawString("X = "+X+"; Y = "+Y, 10, 20); //affichage coordonnées pointeur
        g.drawRect(r.x,r.y,r.width,r.height);
        g.setColor(Color.red);
        int yf,yi=(int)(30*Math.sin(t)); //dessin d'une sinusoïde dans le rectangle
        for (int i=r.x; i<r.x+r.width; i++){
            t+=0.05;
            yf=(int)(30*Math.sin(t));
            g.drawLine(i,z-yi,i+1,z-yf);
            yi=yf;}
        if (actif){ //un bouton est enfoncé
            g.setColor(Color.black); //bouton gauche seul
            if (droit) g.setColor(Color.yellow); //bouton droit
            if (ctrl) g.setColor(Color.cyan); //[[Ctrl] + bouton gauche
            g.drawLine(X,r.y,X,r.y+r.height); //dessin du réticule
            g.drawLine(r.x,Y,r.x+r.width,Y);} }
    public boolean mouseDown(Event evt, int x, int y)
    {
        droit =(evt.modifiers==Event.META_MASK) ? true : false; //test bouton droit
        ctrl =(evt.modifiers==Event.CTRL_MASK) ? true : false; //[[Ctrl] enfoncée
        actif =(r.inside(x,y)) ? true : false; //dans le rectangle ?
        X=x; Y=y; //récupération des coordonnées du pointeur
        repaint(); //pour voir le résultat
        return true;}
    public boolean mouseUp(Event evt, int x, int y)
    {
        actif=false; //si bouton libre => arrêt du tracé du réticule
        repaint(); return true;}
    public boolean mouseDrag(Event evt, int x, int y)
    {
        actif =(r.inside(x,y)) ? true : false;
        X=x; Y=y; repaint(); return true;}
    public boolean mouseMove(Event evt, int x, int y)
    {
        X=x; Y=y; repaint(); return true;}
    public boolean mouseExit(Event evt, int x, int y)
    {
        dedans=false; repaint(); return true;}
    public boolean mouseEnter(Event evt, int x, int y)
    {
        dedans=true; repaint(); return true;}
}

```



Cliquer avec le bouton gauche dans le rectangle

Cliquer avec le bouton droit dans le rectangle

Enfoncer la touche [Ctrl] puis cliquer avec le bouton gauche dans le rectangle

On peut noter que l'affichage vacille (plus la surface de l'applet est importante plus l'effet est marqué). Ceci est lié au fait que l'on oblige l'applet à se redessiner en permanence lors des mouvements de la souris.

La solution est la même que pour les animations : il faut **utiliser un double tampon**.

Dans les variables de classe, il faut déclarer par exemple : `Image ima; Graphics h;`

Dans `init()`, il faut les créer avec : `ima = createImage(size().width,size().height); h = ima.getGraphics();`

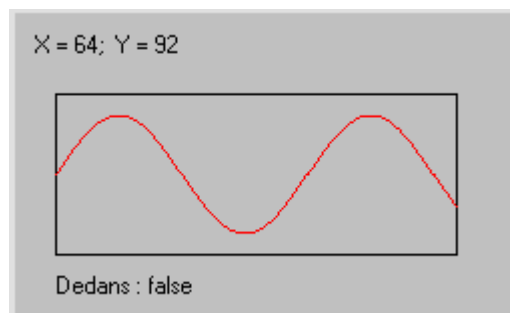
Dans `paint()` il faut remplacer toutes les références à `g` par des références à `h`.

Dans `paint()` ajouter comme première instruction : `h.clearRect(0,0,size().width,size().height);`

Dans `paint()` ajouter comme dernière instruction : `g.drawImage(ima,0,0,Color.white,this);`

Enfin ne pas oublier de surcharger la méthode `public void update(Graphics g){paint(g);}`

La comparaison des deux applets montre dans ce cas l'intérêt de la technique du double tampon.



9.3 Gestion du clavier (Java 1.1.x)

Pour améliorer (en principe) la fiabilité du fonctionnement, la gestion des événements à été profondément modifiée à partir des versions 1.1.0 de JAVA.

Une applet ne peut répondre au clavier que si on a mis en place la classe **KeyListener**. Il faut commencer par importer le package `java.awt.event.*` puis dans l'en-tête ajouter `implements KeyListener`, dans `init()` il faut ajouter `addKeyListener(this)`. L'argument `this` indique que c'est l'applet qui récupère les événements.

Le fait d'ajouter "`implements KeyListener`", implique que l'on surcharge les trois méthodes `keyPressed()`, `keyTyped()` et `keyReleased()` même si ne souhaite pas utiliser les trois.

L'argument de ces méthodes est un objet **KeyEvent**. L'utilisateur peut récupérer le caractère frappé ou le code de la touche grâce aux méthodes `getKeyChar()`, `getKeyCode()` et `getModifiers()` de la classe `KeyEvent`. Il est préférable de réserver `getKeyCode` pour récupérer les codes des touches de fonction. Pour simplifier (?) les noms et les valeurs des constantes des touches ont été modifiées par rapport aux versions 1.0.x.

L'exemple ci-dessous exécute les mêmes tâches que l'applet "`clavier`" du § 9.1. Les principales modifications du code sont indiquées en rouge.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class clavier11 extends Applet implements KeyListener
{ char touche='@';
  int taille=20,code=touche,gras=0,c=125;
  boolean maj;
  public void init()
  { addKeyListener(this);
    setBackground(Color.lightGray);
    this.requestFocus();}
  public void paint(Graphics g) //inchangée par rapport au code de la version 1.0
  { Color col = new Color(c,0,0);
    Font font = new Font("Helvetica",gras,taille);
    g.setFont (font);    g.setColor(col);
    g.drawString("taille = "+taille, 10, 60);
    g.drawString("Maj = "+maj, 10, 110);
    g.drawString(String.valueOf(touche)+" => "+code,10,170);}
  public void keyReleased(KeyEvent evt) //ici vide mais obligatoire
  { }
  public void keyTyped(KeyEvent evt)
  { touche=evt.getKeyChar();} //caractère
  public void keyPressed(KeyEvent evt)
  { int t=evt.getKeyCode(); //code touche
    int m=evt.getModifiers(); //masque modificateurs clavier
    switch(t){
      case evt.VK_DOWN : if (taille>6) taille--; break;
      case evt.VK_UP : if (taille<55) taille++; break;
      case evt.VK_LEFT : if (c>0) c-=5; break;
      case evt.VK_RIGHT : if (c<255) c+=5; break;
      case evt.VK_PAGE_UP : gras=1; break;
      case evt.VK_PAGE_DOWN : gras=0; break;}
    code=t; touche=(char)27;
    if ((m & evt.SHIFT_MASK) != 0) maj=true; else maj=false;
    repaint();}
}
```

taille = 22

Maj = false

□ => 17

9.4 Gestion de la souris (Java 1.1.x)

Une applet ne peut répondre aux événements "souris" que si la classe `MouseListener` (à l'écoute de la souris) est mise en service par l'ajout dans son en-tête de `implements MouseListener`. (Il faut au préalable importer le package `java.awt.event.*`). Ceci suppose que l'on surcharge les méthodes `mousePressed()`, `mouseReleased()`, `mouseEntered()`, `mouseExited()` et `mouseClicked()`. Il faut surcharger ces 5 fonctions même si seulement quelques unes sont en fait utilisées. L'argument de ces fonctions est un objet **MouseEvent** dont les méthodes permettent d'obtenir les informations sur les coordonnées du pointeur, l'objet à l'origine de l'événement ...

Si l'on désire obtenir des informations sur les mouvements de la souris, il faut aussi mettre en oeuvre la classe `MouseMoveListener`. Il faut alors implémenter les deux méthodes `mouseMoved()` et `mouseDragged()` dont l'argument est également un objet `MouseEvent`. La méthode `init()` de l'applet active les "écouteurs" par l'appel aux méthodes `addMouseListener(this)` et `addMouseMoveListener(this)`.

Alors que dans les versions 1.0.x, il était pratiquement impossible de modifier la forme du pointeur de la souris, à partir de la version 1.1.0, c'est devenu très simple. Il suffit de déclarer un objet **Cursor** avec l'instruction `Cursor nom = new Cursor(int c)`; il existe une série prédéfinie de curseurs. Par exemple la constante `Cursor.HAND_CURSOR` correspond au curseur en forme de main. L'instruction `setCursor(nom)` provoque l'affichage de ce pointeur.

L'exemple ci-dessous exécute les mêmes tâches que l'applet "souris" du § 9.2 (version avec double tampon). Les principales modifications du code sont indiquées en rouge.

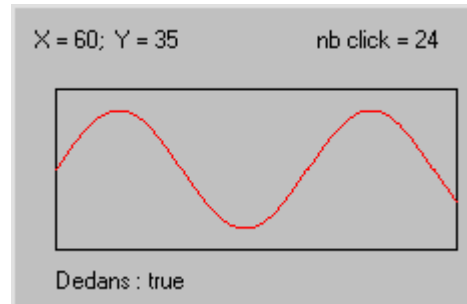
```

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class souris11 extends Applet implements MouseMotionListener,MouseListener
{
    Font font;
    Rectangle r=new Rectangle(20,40,200,80);
    Image ima; Graphics h; //double tampon
    int la,ha,X,Y,z=r.y+40,nb;
    boolean actif,droit,dedans,ctrl;
    Cursor main = new Cursor(Cursor.HAND_CURSOR);
    Cursor norm = new Cursor(Cursor.DEFAULT_CURSOR);
    public void init()
    {
        setBackground(Color.lightGray);
        font = new Font("Helvetica",0,11);
        la=getSize().width; ha=getSize().height; //remplace size() de 1.0.x
        ima=createImage(la,ha); h=ima.getGraphics();
        addMouseMotionListener(this);
        addMouseListener(this);
    }
    public void update(Graphics g) // double tampon
    {
        paint(g);
    }
    public void paint(Graphics g // identique à la version 1.0
    {
        double t=0;
        h.clearRect(0,0,la,ha);
        h.setColor(Color.black);
        h.drawString("Dedans : "+dedans,20,140);
        h.drawString("X = "+X+"; Y = "+Y, 10, 20);
        h.drawString("nb click = "+nb, 150, 20);
        h.drawRect(r.x,r.y,r.width,r.height);
        h.setColor(Color.red);
        int yf,yi=(int)(30*Math.sin(t));
        for (int i=r.x; i<r.x+r.width; i++){
            t+=0.05; yf=(int)(30*Math.sin(t));
            h.drawLine(i,z-yi,i+1,z-yf); yi=yf;}
        if (actif){
            h.setColor(Color.black);
            if (droit) h.setColor(Color.yellow);
            if (ctrl) h.setColor(Color.cyan);
            h.drawLine(X,r.y,X,r.y+r.height);
            h.drawLine(r.x,Y,r.x+r.width,Y);}
        g.drawImage(ima,0,0,Color.white,this);}
    public void mouseReleased(MouseEvent evt) //bouton relâché
    {
        actif= false;
        setCursor(norm);
        repaint();
    }
    public void mousePressed(MouseEvent evt) //bouton pressé
    {
        int m=evt.getModifiers();
        setCursor(main);
        droit=(m == evt.META_MASK) ? true : false; //touche Meta ou bouton droit
        ctrl=((m & evt.CTRL_MASK)!=0) ? true : false; //touche CTRL
        X=evt.getX(); Y=evt.getY(); //coordonnées du pointeur
        actif =(r.contains(X,Y)) ? true : false; //remplace inside de 1.0.x
        repaint();
    }
    public void mouseEntered(MouseEvent evt) //entrée du pointeur dans le cadre
    {
        dedans=true; repaint();
    }
    public void mouseExited(MouseEvent evt) //sortie du cadre
    {
        dedans=false; repaint();
    }
}

```

```
public void mouseClicked(MouseEvent evt) //pressé puis relaché
{ nb++; repaint();}

public void mouseMoved(MouseEvent evt) //déplacement bouton relaché
{ X=evt.getX(); Y=evt.getY(); repaint();}
public void mouseDragged(MouseEvent evt) //déplacement bouton pressé
{ X=evt.getX(); Y=evt.getY();
  actif =(r.contains(X,Y) ? true : false;
  repaint();}
}
```



10. Gestionnaires de mise en page

Tous les systèmes d'exploitation moderne sont maintenant dotés d'interfaces graphiques. Pour assurer l'indépendance de JAVA par rapport à la plate-forme, les concepteurs auraient pu concevoir des composants graphiques spécifiques. Ils ont préféré réutiliser les composants de la machine client au moyen d'interfaces implantés dans la machine virtuelle du navigateur. La gestion des composants graphiques est basée sur la notion de **conteneurs** qui sont des espaces graphiques destinés à recevoir plusieurs composants comme des boutons, des cases à cocher ... dont la mise en place graphique est fonction d'un **protocole de mise en page**. En effet, selon les systèmes d'exploitation, la taille et l'aspect de ces composants varient et il se pose le problème de leur positionnement. La classe **Applet** dérive de la classe **Panel** qui est une extension de la classe abstraite **Container**. La classe **Panel** (panneau) représente des conteneurs qui sont placés dans un espace graphique existant pouvant être un autre **Panel** : une applet étant un **Panel** peut donc contenir d'autres **Panels** (système récursif).

Avant d'examiner au chapitre suivant les composants graphiques gérés par JAVA, nous allons maintenant examiner les différents gestionnaires de mise en page associés aux conteneurs.

10.1 FlowLayout

C'est la **mise en page par défaut** (elle est utilisée quand aucun protocole n'est précisé). Dans cette mise en page "glissante", les composants graphiques sont ajoutés les uns après les autres de gauche à droite avec un saut de ligne quand il n'y a plus de place à droite. L'aspect final dépend donc de la taille de la fenêtre de l'applet.

Les trois constructeurs possibles possèdent la syntaxe :

FlowLayout(), **FlowLayout(int align)** et **FlowLayout(int align, int dx, int dy)**. *dx* et *dy* précisent l'écartement des composants en *x* et *y*.

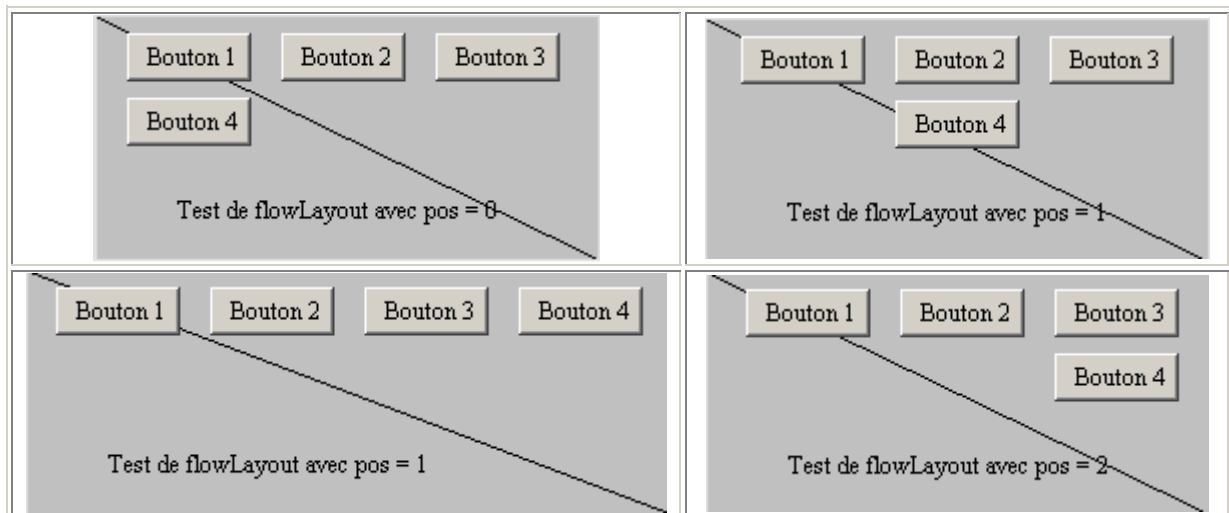
Les constantes *FlowLayout.LEFT* = 0, *FlowLayout.CENTER* = 1 (défaut) et *FlowLayout.RIGHT* = 2 précisent l'alignement (*align*) dans une ligne.

Dans l'exemple suivant, on utilise comme composants un tableau de 4 boutons. La taille des boutons est fonction de la fonte utilisée : il est donc essentiel de définir celle-ci. La valeur de la constante *pos* est passée à l'applet par une balise <param>. On peut constater l'évolution de l'aspect final en fonction de la taille de l'applet. Remarquez aussi que seule la surface des composants masque la zone de dessin.

```

import java.applet.*;
import java.awt.*;
public class layflow extends Applet
{ Button bt[] = new Button[4]; //déclaration des boutons
  Font font= new Font("TimesRoman",0,12);
  FlowLayout fl; //déclaration du gestionnaire
  int p;
  public void init()
  { setBackground(Color.lightGray);
    setFont(font);
    p = Integer.parseInt(getParameter("pos")); //récupération de pos
    fl = new FlowLayout(p,15,8); //création du gestionnaire
    setLayout(fl); //mise en place
    for (int i=0; i<5; i++){
      bt[i]=new Button(" Bouton "+(i+1)); //création des boutons
      add(bt[i]);} // méthode add( ) pour la mise en place des boutons
  public void paint(Graphics g)
  { g.setFont(font);
    g.drawString("Test de flowLayout avec pos = "+p, 40, 100);
    g.drawLine(0,0,size().width,size().height);}
  }

```



Dans tous les cas de mise en place simple avec peu de composants il faut utiliser ce gestionnaire.

10.2 BorderLayout

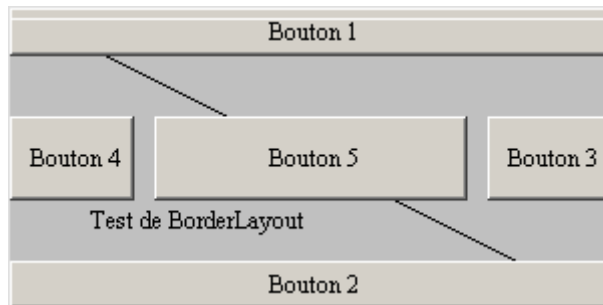
La mise en place glissante offre peu de possibilité pour les mises en page complexes. Le protocole de spécification par les bords **BorderLayout** décompose le conteneur en 5 zones (précisées par les constantes North, South, East, West et Center) qui peuvent recevoir chacune **un seul** composant. Le constructeur **BorderLayout(int dx, int dy)** permet de préciser l'espace entre les composants selon x et y. Les dimensions des composants sont fonctions des dimensions du container mais leurs positions sont maintenant définies. L'exemple ci-dessous indique comment utiliser ce gestionnaire.

```

import java.applet.*;
import java.awt.*;
public class layborder extends Applet
{ Button bt[]=new Button[5];
  Font font= new Font("TimesRoman",0,12);
  BorderLayout fl;
  public void init()
  { setBackground(Color.lightGray);
    setFont(font);
    fl = new BorderLayout(10,30);//création du gestionnaire
    setLayout(fl);
    for (int i=0; i<5; i++)
      bt[i]=new Button(" Bouton "+(i+1));//création des boutons
    add("North",bt[0]);  add("South",bt[1]);//mise en place
    add("East",bt[2]);  add("West",bt[3]);
    add("Center",bt[4]);  }

  public void paint(Graphics g)
  { g.setFont(font);
    g.drawString("Test de BorderLayout",40,110);
    g.drawLine(0,0,size().width,size().height);}
}

```



Avec cet exemple simpliste ce protocole semble peu intéressant puisque chaque composant graphique occupe toute la place qui lui est offerte. C'est ici que l'on peut faire intervenir la récursivité des protocoles : **un conteneur est lui-même un composant graphique qui peut contenir d'autres conteneurs**.

Le conteneur le mieux adapté est le panneau (**Panel**). Avec cette technique, il est possible de concevoir des mises en page d'applets assez complexes avec un code relativement simple. Ainsi dans l'exemple suivant, nous allons placer des composants dans le haut et dans le bas de l'applet en utilisant deux objets panneaux dotés chacun de son propre gestionnaire de mise en page. Pour la mise en place des boutons dans chaque panneau, on utilise la méthode add() de l'instance de Panel utilisée.

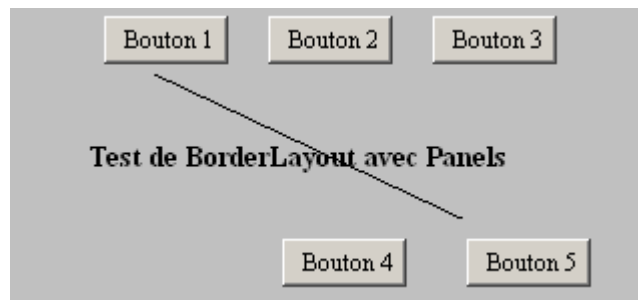
```

import java.applet.*;
import java.awt.*;
public class layborder2 extends Applet
{ Button bt[]=new Button[5];
  Font font= new Font("TimesRoman",0,12);
  Font bold= new Font("TimesRoman",1,14);
  Panel Panor=new Panel(), Pasud=new Panel(); //création des Panels
  BorderLayout fl= new BorderLayout(10,30); //Protocole principal

public void init()
{ setBackground(Color.lightGray);
  setFont(font); //fonte pour les composants
  setLayout(fl);
  add("North",Panor); add("South",Pasud); //ajout des panneaux
  Panor.setLayout(new FlowLayout(1,20,5)); //protocole du panneau supérieur
  Pasud.setLayout(new FlowLayout(2,30,10)); //protocole du panneau inférieur
  for (int i=0; i<5; i++) //création des boutons
    bt[i]=new Button(" Bouton "+(i+1));
  for (int i=0; i<3; i++) Panor.add(bt[i]); //mise en place des boutons du haut
  for (int i=3; i<5; i++) Pasud.add(bt[i]);}

public void paint(Graphics g)
{ g.setFont(bold); //fonte pour l'applet
  g.drawString("Test de BorderLayout avec Panels",40,80);
  g.drawLine(0,0,size().width,size().height);}
}

```



Attention : Les Panels ainsi mis en place limitent la surface de dessin de l'applet !

10.3 GridLayout

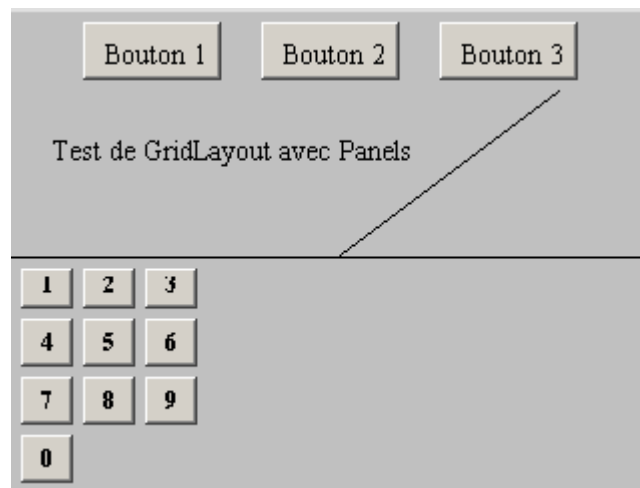
Ce protocole définit une grille dont chaque cellule peut contenir un composant graphique. Les composants sont ajoutés de gauche à droite ligne par ligne. Utilisé seul ce gestionnaire n'a pas beaucoup d'intérêt. Par contre utilisé avec un protocole BorderLayout et des panneaux, il permet des mises en page intéressantes. Il existe deux constructeurs **GridLayout(int lig, int col)** et **GridLayout(int lig, int col, int dx, int dy)**; *lig* indique le nombre de lignes de la grille et *col* son nombre de colonnes; *dx* et *dy* précisent l'écartement entre les composants.

Dans l'exemple suivant, on met en place dans un panneau Pasud un autre panneau PasuW doté d'un protocole Grid Layout qui va contenir 10 boutons. Il est en effet impossible de placer directement ce protocole dans Pasud car alors les composants occuperaient toute la largeur offerte. La surface en dessous du trait horizontal n'est plus disponible pour le dessin (elle est occupée par le panneau Pasud) mais elle peut éventuellement contenir des composants.

```

import java.applet.*;
import java.awt.*;
public class laygrid extends Applet
{ Button bt[]=new Button[3];
  Button bu[]=new Button[10];
  Font font= new Font("TimesRoman",0,14);
  Font bold= new Font("TimesRoman",1,12);
  Panel Panor=new Panel(), Pasud=new Panel();
  Panel PasuW=new Panel(); //containeur pour les boutons bu
  BorderLayout fl= new BorderLayout(10,10);
  public void init()
  { setBackground(Color.lightGray);
    Panor.setFont(font); Pasud.setFont(bold); //choix des fontes
    setLayout(fl);
    add("North",Panor); add("South",Pasud);
    Panor.setLayout(new FlowLayout(1,20,5)); //protocoles des panneaux
    Pasud.setLayout(new FlowLayout(0,5,5)); //principaux
    Pasud.add(PasuW); //1 panneau dans un autre
    PasuW.setLayout(new GridLayout(4,3,5,5));
    //avec son gestionnaire (une grille de 4 lignes de 3 colonnes)
    for (int i=0; i<3; i++) //création des boutons bt
      bt[i]=new Button(" Bouton "+(i+1));
    for (int i=0; i<10; i++) //création des boutons bu
      bu[i]=new Button(" "+i+" ");
    for (int i=0; i<3; i++) Panor.add(bt[i]); //et mise en place
    for (int i=1; i<10; i++) PasuW.add(bu[i]);
    PasuW.add(bu[0]);}
  public void paint(Graphics g)
  { g.setFont(font);
    g.drawString("Test de GridLayout avec Panels",20,70);
    g.drawLine(0,118,320,118); //pour contrôle de la zone utile
    g.drawLine(0,size().height,size().width,0);}
}

```



10.4 Autres protocoles

Il existe également le gestionnaire **CardLayout** qui ne peut afficher qu'un seul composant à la fois (en général un panneau qui va posséder ses propres composants). Ce gestionnaire est doté de méthodes `first()`, `last()`, `previous()` et `shows()` qui permettent de naviguer entre les cartes. Ceci suppose la mise en place de boutons de navigation sur chaque carte.

Enfin pour les situations complexes, existe le protocole **GridBagLayout** qui fonctionne avec des contraintes précisées par la classe **GridBagConstraints**. Dans ce gestionnaire chaque composant est inséré dans un conteneur et on lui associe une contrainte qui précise son emplacement dans une grille, sa taille...

Ce protocole est très lourd à mettre en oeuvre et je ne le décrirai pas car il existe (à mon avis) une méthode beaucoup plus facile à mettre en application dans les applets qui est la méthode sans gestionnaire.

10.5 Mise en page sans gestionnaire

Dans une applet, la taille de la fenêtre est imposée par les paramètres `width` et `height` de la balise `<applet>`. Il est donc possible de prévoir l'aspect final de l'applet dans le navigateur si la taille des composants est connue. Il existe pour les composants la méthode **reshape(int ox, int oy, int large, int haut)** qui permet d'imposer au composant sa position (`ox`, et `oy`) et ses dimensions (`large` et `haut`) à condition qu'un gestionnaire de mise en page ne soit pas actif.

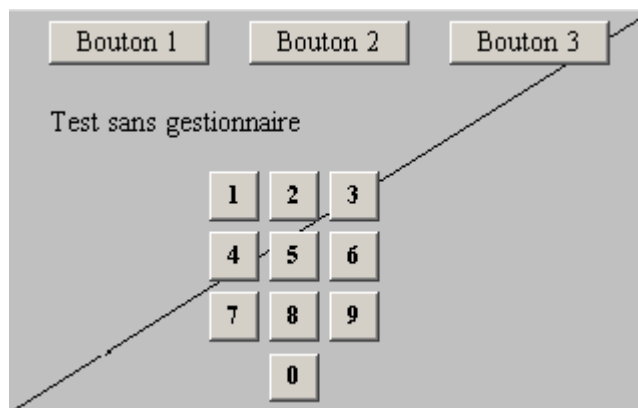
Comme par défaut JAVA utilise un protocole `FlowLayout`, il faut commencer par déclarer que celui-ci est inactif au moyen de l'instruction **FlowLayout fl = null**. On peut ensuite placer les composants dans la fenêtre aux endroits souhaités. Cette technique suppose que la fonte utilisée pour les composants soit définie dès le départ. Comme exemple, on va reprendre le cas précédent. On peut constater que cette technique est beaucoup plus souple au niveau du positionnement des composants. On peut aussi constater qu'aucun panneau ne vient masquer la surface de l'applet.

Remarque : depuis la version 1.1, la méthode **reshape()** a été remplacée par la méthode **setBounds()** qui possède les mêmes arguments (en fait seul le nom de la méthode a changé).

```

import java.applet.*;
import java.awt.*;
public class laynull extends Applet
{ Button bt[]=new Button[3];
  Button bu[]=new Button[10];
  Font font= new Font("TimesRoman",0,14);
  Font bold= new Font("TimesRoman",1,12);
  BorderLayout fl= null; //essentiel
public void init()
{ setBackground(Color.lightGray);
  setLayout(fl); //essentiel
  for (int i=0; i<3; i++){
    bt[i]=new Button("Bouton "+(i+1));//création
    add(bt[i]); //ajout
    bt[i].setFont(font); //fonte du composant
    bt[i].reshape(20+100*i,5,80,22); //mise en place
  setFont(bold); //fonte par défaut pour les composants à venir
  for (int i=0; i<10; i++)
    bu[i]=new Button(" "+i+" ");//création
  int k=0;
  for (int l=80; l<150; l+=30){
    for (int c=0; c<3; c++){
      k++;
      add(bu[k]); //ajout
      bu[k].reshape(100+30*c,l,25,25); //mise en place
    }
  }
  add(bu[0]);
  bu[0].reshape(130,170,25,25);}
public void paint(Graphics g)
{ g.setFont(font);
  g.drawString("Test sans gestionnaire",20,60);
  g.drawLine(0,size().height,size().width,0);}
}

```



11. Les composants graphiques (1.0)

Tous les systèmes d'exploitation modernes sont maintenant dotés d'interfaces graphiques. Au lieu de concevoir des composants spécifiques à JAVA, les concepteurs du langage ont préféré réutiliser les composants de la machine client au moyen d'interfaces implantés dans la machine virtuelle du navigateur ce qui assure l'indépendance de JAVA par rapport à la plateforme. Le positionnement de ces composants dans la fenêtre de l'applet est déterminé par les protocoles de mise en page déjà étudiés au chapitre 10. La majorité des composants répondent aux actions des utilisateurs (click souris, touche [Enter] ...) par l'envoi d'un **événement** au conteneur du composant. Dans les versions 1.0.x ces événements peuvent être interceptés et traités par une méthode **handleEvent(Event evt)** de la classe **Component**. Si l'événement n'est pas traité par ce conteneur, il remonte au conteneur qui le contient : en général pour les applets, il suffit d'implémenter la méthode **handleEvent()** de l'applet puisque c'est elle qui est le conteneur final de tous les composants. Pour les événements simples la méthode **handleEvent()** appelle la méthode **action(Event evt, Object arg)**. La variable *evt.target* permet de récupérer la nature du composant à l'origine de l'événement.

Pour **tous** les composants, il est possible de préciser lors de la création la fonte associée au texte et la couleur du pinceau au moyen de la méthode **setForeground()**. Il est possible de cacher un composant de nom *cp* par la méthode **cp.hide()**, de le réafficher avec **cp.show()**, de le désactiver avec **cp.disable()** et de le réactiver avec **cp.enable()**.

11.1 Classe Label

Les libellés sont des objets de la classe **Label**. Ils permettent d'afficher un texte dans un conteneur. Le texte d'un Label peut être modifié librement mais la longueur du nouveau texte ne doit pas excéder celle définie lors de la mise en place par la méthode **add()** dans le gestionnaire de mise en page.

Ce composant ne génère pas d'action.

Les paramètres d'alignement sont **Label.LEFT = 0**, **Label.CENTER = 1** et **Label.RIGHT = 2**. Les constructeurs sont **Label()**, (Label sans texte), **Label(String s)**, (Label de texte *s*) et **Label(String s, int aligne)** qui précise l'alignement dans le conteneur.

Les méthodes d'instances d'un Label de nom "lb" sont **lb.getText()** qui permet de récupérer le libellé du Label et **lb.setText(String s)** qui permet de le modifier.

11.2 Classe Button

Les boutons appartiennent à la classe **Button**. Ce sont des rectangles portant un titre et dont l'aspect se modifie lorsque l'on clique sur eux (inversion de l'effet de relief). Ils émettent alors l'événement **ACTION_EVENT**.

Les constructeurs sont **Button()**, (Bouton sans texte) et **Button(String label)** (Bouton avec le texte *label*).

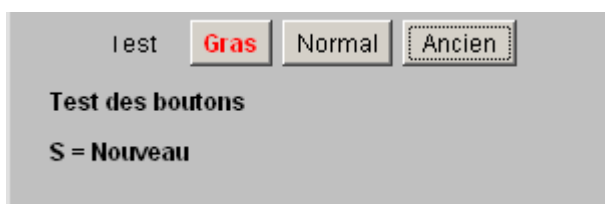
Ce sont les dimensions de la chaîne *label* initiale qui définissent les dimensions finales du bouton : il est donc important de commencer par préciser la fonte utilisée avant de placer le bouton.

Les méthodes d'instances d'un bouton de nom "bt" sont **bt.getLabel()** qui permet de récupérer le libellé du bouton et **bt.setLabel(String s)** qui permet de le modifier avec les mêmes restrictions pour les dimensions que pour les Labels.

L'exemple suivant met en oeuvre les méthodes des libellés et de boutons. Pour ne pas surcharger le listing, il n'est pas précisé de protocole de mise en page : c'est donc le gestionnaire par défaut (FlowLayout en mode centré) qui est utilisé. On trouvera aussi un exemple d'utilisation de la méthode `action()` pour le contrôle des événements. Les composants sont identifiés à partir des événements mais il est aussi possible d'utiliser le paramètre `arg` qui retourne ici le libellé. On modifie le libellé du troisième bouton à chaque click. Il faut prévoir pour son texte initial une longueur suffisante afin que le texte de remplacement puisse être affiché en totalité. L'exemple indique également comment utiliser les méthodes `show()`, `hide()`, `enable()` et `disable()`.

```
import java.applet.*;
import java.awt.*;
public class boutons extends Applet
{
    Font font = new Font("Helvetica",0,12);
    Font bold = new Font("Helvetica",1,12);
    Label lb = new Label("Test"); //déclaration et création
    Button bt1, bt2, bt3;
    boolean gras;    String s;
public void init()
{
    setBackground(Color.lightGray);
    setFont(font); //fonte par défaut pour les composants et l'applet
    add(lb);        //ajout du Label par le gestionnaire
    bt1 = new Button("Gras"); //création
    bt1.setForeground(Color.red); //couleur du texte du bouton
    bt1.setFont(bold);    add(bt1); //fonte spécifique et ajout
    bt2 = new Button("Normal");
    add(bt2);
    bt3 = new Button(" Ancien ");
    add(bt3);}

public boolean action(Event evt, Object arg)
{
    s = arg.toString();
    if (evt.target.equals(bt1)) gras = true;
    else if (evt.target==bt2) gras = false; //autre syntaxe possible
    else if (evt.target.equals(bt3)){
        if (bt3.getLabel()==" Ancien "){
            bt3.setLabel("Nouveau");//modification du libellé à chaque click
            bt1.hide(); //cacher bt1
            bt2.disable();} //désactiver bt2
        else {
            bt3.setLabel(" Ancien ");
            bt1.show();
            bt2.enable();}}
    else return super.action(evt, arg); //pour événements non traités ici
    repaint();
    return true;}
public void paint(Graphics g)
{
    if (gras) g.setFont(bold); else g.setFont(font);
    g.drawString("Test des boutons", 20, 50);
    g.drawString(s, 20, 65);}
}
```



Tester les 3 boutons

11.3 Classe Checkbox

Il existe deux types de cases à cocher : les cases à choix multiples qui sont indépendantes les unes des autres et les cases à choix exclusif nommées aussi boutons radio. Dans un groupe à choix exclusif, une seule case peut être cochée à la fois. Il alors faut inclure les cases à cocher dans un groupe grâce à un objet *CheckboxGroup*.

Les cases à cocher possèdent deux états : cochées ou non cochées. Un événement *ACTION_EVENT* est généré à chaque changement d'état.

Les constructeurs des cases à choix multiple sont *Checkbox()* (case sans label) et *Checkbox(String label)* (case avec un libellé);

Pour les boutons radio, le constructeur est *Checkbox(String label, CheckboxGroup nom_groupe, boolean etat)*; il associe la case au groupe *nom_groupe* et précise son état (cochée ou non).

On récupère l'état (booléen) de la case de nom *cb* avec la méthode **cb.getState()** et on l'impose avec la méthode **cb.setState(boolean e)**.

Dans l'exemple suivant les cases *cb1* et *cb2* sont des cases à choix multiple (les deux peuvent être cochées ou décochées en même temps). Les autres cases forment un groupe exclusif : une seule case est cochée. La méthode *init()* indique comment ajouter les composants dans le protocole de mise en page (Ici le gestionnaire. par défaut : *FlowLayout* en mode centré).

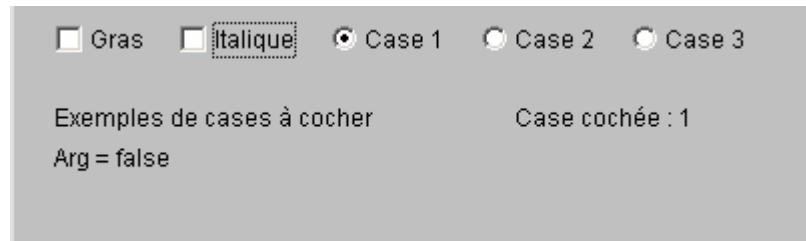
La méthode *action()* donne une façon possible de traiter les événements.

```
import java.applet.*;
import java.awt.*;
public class boxcheck extends Applet
{
    Font font = new Font("Helvetica",0,12);
    Checkbox cb1,cb2,cb3,cb4,cb5; //déclarations des cases
    CheckboxGroup cg = new CheckboxGroup(); //déclarations du groupe
    boolean gras,ital;
    int index=1;      String s,s1;
public void init()
{
    setBackground(Color.lightGray);
    setFont(font);
    cb1 = new Checkbox("Gras");      add(cb1); //création et ajout
    cb2 = new Checkbox("Italique"); add(cb2);
    cb3 = new Checkbox("Case 1",cg,true); //création bouton radio
    add(cb3);
    cb4 = new Checkbox("Case 2",cg,false);      add(cb4);
    cb5 = new Checkbox("Case 3",cg,false);      add(cb5);}
public boolean action(Event evt, Object arg)
{
    s1=arg.toString();
    if (evt.target==cb1) {
        gras = (cb1.getState()) ? true : false;} //test de l'état de la case
    else if (evt.target==cb2) {
        ital = (cb2.getState()) ? true : false;}
    else if (evt.target==cb3) index=1; //choix exclusif
    else if (evt.target==cb4) index=2;
    else if (evt.target==cb5) index=3;
    else return super.action(evt,arg);
    repaint();
    return true;}
}
```

```

public void paint(Graphics g)
{
    int mode=0;
    if (gras) mode+=1;
    if (ital) mode+=2; //ajout des valeurs si les deux cases sont cochées
    font = new Font("Helvetica",mode,12);
    g.setFont(font);
    g.drawString("Exemples de cases à cocher",20,60);
    g.drawString("Case cochée : "+index,250,60);
    g.drawString("Arg = "+s1,20,80);}
}

```



11.4 Classes Choice et List

La classe Choice correspond à des menus déroulants. Au repos, ce composant affiche l'item sélectionné. Après un click, il affiche l'ensemble des items du menu et il devient alors possible de faire une nouvelle sélection par un click qui génère un événement ACTION_EVENT. Après création de la liste *ch* par le constructeur *Choice()*, il faut ajouter chaque item de la liste avec la méthode *ch.addItem(String label)*. La position de l'item sélectionné (le premier index vaut 0) est donnée par la méthode *ch.getSelectedIndex()*. La méthode *ch.getSelectedItem()* retourne le libellé de l'item sélectionné. Il est possible de supprimer l'item d'index *p* avec *ch.removeItem(int p)* ou de le remplacer par *ch.addItem(String label, int p)*.

Attention : Le déroulement d'un menu Choice masque une partie de l'applet. Dans certains navigateurs la fermeture du menu déclenche le rafraîchissement de l'affichage par un appel à *repaint()*.

La classe Liste correspond à des listes de choix. Au repos, ce composant affiche plusieurs items. Un ascenseur permet de visualiser (si nécessaire) la totalité de la liste. Comme la liste peut autoriser des choix multiples la sélection d'un item par un simple click génère un événement LIST_SELECT qui doit être traité par la méthode *handleEvent()*. De même la désélection d'un item génère un événement LIST_DESELECT. Par contre la sélection d'un item par un double click génère un événement ACTION_EVENT.

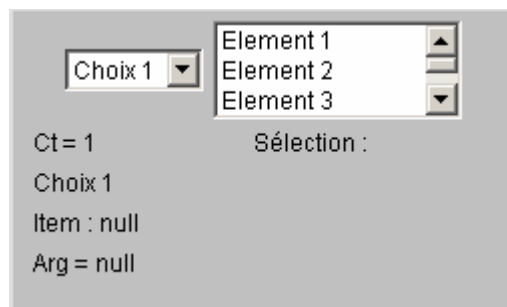
Les constructeurs sont *List()* (liste simple) et *List(int nb, boolean chMul)* qui définit une liste affichant *nb* items et qui autorise le choix multiple si le booléen *chMul* est vrai. Les méthodes de création des items et récupération de l'index sont les mêmes que pour la classe Choice. La méthode *getSelectedItems()* retourne un tableau de chaînes qui contient les éléments sélectionnés. Ne pas oublier le s à la fin du nom de la méthode.

L'exemple suivant montre comment utiliser les listes. Le compteur *ct* indique le nombre de passage dans la méthode *paint()*. Ce compteur permet de vérifier si l'utilisation du menu déroulant provoque un *repaint()* non demandé par le programmeur. (fonction du navigateur utilisé)

```

import java.applet.*;
import java.awt.*;
public class liste extends Applet
{ Font font = new Font("Helvetica",0,12);
  Font bold = new Font("Helvetica",1,12);
  Choice choix;
  List liste;
  int nch,nliste,ct;
  String s,sl,sel[]=new String[5];
public void init()
{ setBackground(Color.lightGray);
  setFont(font);
  choix = new Choice();
  choix.addItem("Choix 1");      choix.addItem("Choix 2");
  choix.addItem("Choix 3"); //création des items
  add(choix); //ajout par le protocole de mise en page
  liste = new List(3,true); //affichage sur 3 lignes, multiple autorisé
  liste.addItem("Element 1");   liste.addItem("Element 2");
  liste.addItem("Element 3");   liste.addItem("Element 4");
  liste.addItem("Element 5");
  add(liste);}
public boolean action(Event evt, Object arg)
{ sl=arg.toString(); //retour de la chaîne de l'item sélectionné
  if (evt.target==choix) {
    nch=choix.getSelectedIndex();//index de l'item sélectionné
    s=choix.getSelectedItem();
    if (nch==2) ct=0;} //RAZ du compteur de passage dans paint()
  else if (evt.target.equals(liste))
    sel=liste.getSelectedItems();
  else return super.action(evt,arg);
  repaint();
  return true;}
public void paint(Graphics g)
{ ct++; g.drawString("Ct = "+ct,10,70);
  g.drawString("Choix "+(nch+1),10,90);
  g.drawString("Item : "+s,10,110);
  g.drawString("Sélection : ",120,70);
  for (int i=0; i<sel.length; i++)
    g.drawString(sel[i],120,85+15*i);
  g.drawString("Arg = "+sl,90,110);}
}

```



Double click sur la liste

11.5 Classe TextField

C'est une zone de texte sur une ligne. Le contenu du champ est éditable par l'utilisateur. La touche "retour chariot" [Enter] génère un événement ACTION_EVENT. Les constructeurs sont *TextField()* qui retourne un champ vide, *TextField(int n)* qui crée un champ vide de n caractères, *TextField(String s)* qui crée un champ initialisé avec la chaîne s et *TextField(String s, int n)*. Les principales méthodes que l'on peut appliquer sur une zone de texte de nom *tf* sont *tf.getText()* qui retourne la chaîne affichée et *tf.setText(String ns)* qui affiche la chaîne *ns* dans le champ. Pour faire des saisies avec un masque d'écriture, on peut aussi utiliser la méthode *tf.setEchoCharacter(char c)*. La classe *TextArea* (zone de texte sur plusieurs lignes) possède des méthodes analogues.

L'exemple montre comment utiliser les zone de texte. Il précise comment effectuer les conversions des chaînes de caractères récupérées vers des entiers ou des réels. Avec ce code, il y a générations d'exceptions si la chaîne tapée dans la zone ne représente pas un nombre valide.

Note : par suite d'un bug (?), le compilateur utilisé génère un code non fonctionnel si les zones de texte sont créées dans la procédure *init()* et pas lors de la déclaration.

```
import java.applet.*;
import java.awt.*;
public class zonetext extends Applet
{ int i=10; double d=5.25;
  String s,s1,sm="*****";
  Font font = new Font("Helvetica",0,12);
  Label lb1,lb2,lb3;
  TextField tf1=new TextField("10",5);
  TextField tf2=new TextField(""+d,5);
  TextField tf3=new TextField(sm,6);

public void init()
{ setBackground(Color.lightGray);
  setFont(font);
  lb1=new Label("Entier",0);
  add(lb1); add(tf1);
  lb2=new Label("Réel",0);
  add(lb2); add(tf2);
  lb3=new Label("Masqué",0);
  add(lb3); add(tf3);
  tf3.setEchoCharacter('*');

public boolean action(Event evt, Object arg)
{ s1=arg.toString();
  if (evt.target==tf1) {
    s=tf1.getText();
    i=Integer.parseInt(s);}//conversion chaîne en entier
  else if (evt.target==tf2) {
    s=tf2.getText();
    d=Double.valueOf(s).doubleValue();}//conversion chaîne en double
  else if (evt.target==tf3){
    sm=tf3.getText();}
  else return super.action(evt,arg);
  repaint(); return true;}
public void paint(Graphics g)
{ g.drawString("Entier = "+i,20,60);
  g.drawString("Double = "+d,120,60);
  g.drawString("Masqué = "+sm,250,60);
  g.drawString("Arg = "+s1,20,80);}
}
```

Entier	<input type="text" value="10"/>	Réel	<input type="text" value="5.25"/>	Masqué	<input type="text" value="*****"/>
Entier = 10		Double = 5.25		Masqué = *****	
Arg = null					

11.6 Classe Scrollbar

A cause des problèmes de mise en page, c'est un composant délicat à mettre en oeuvre car il veut occuper le maximum de place mis à sa disposition. Il faut, surtout si on utilise des ascenseurs verticaux, empiler des panneaux et des gestionnaires pour obtenir un aspect à peu près correct. La mise en page sans protocole est toujours beaucoup plus facile à mettre en application dans le cas des ascenseurs.

Le constantes `Scrollbar.HORIZONTAL = 0` et `Scrollbar.VERTICAL = 1` définissent l'orientation de l'ascenseur.

Le constructeur d'un ascenseur est `Scrollbar(int orient, int valeur, int delta, int mini, int maxi)`. *valeur* est la valeur initiale associée à la position du curseur, *delta* est l'incrément dont varie la valeur quand on clique de part et d'autre du curseur (un click sur les boutons des extrémités fait varier la valeur de une unité), *mini* et *maxi* sont les valeurs extrêmes entre lesquelles peut varier *valeur*. Un ascenseur ne génère pas d'événement `ACTION_EVENT`. Il faut utiliser la méthode `handleEvent()` pour récupérer les événements de ce composant. La méthode la plus utile est `getValue()` qui retourne la valeur liée à la position du curseur.

L'exemple suivant montre comment utiliser les ascenseurs. L'ascenseur vertical est placé à gauche. Il s'étale jusqu'en bas de l'applet. Les deux ascenseurs horizontaux sont placés dans des panneaux imbriqués dans un autre panneau (le gestionnaire `BorderLayout` *bl* admet *un seul* composant dans la case Nord). On peut constater que leur longueur est liée à celle du libellé placé juste au-dessus.

J'ai constaté que comme valeur maximale, il faut prendre la valeur souhaitée augmentée de la valeur de l'incrément delta. (avec un *delta* de 5, il faut prendre *mini* = 0, *maxi* = 105 pour que *valeur* varie entre 0 et 100.)

```

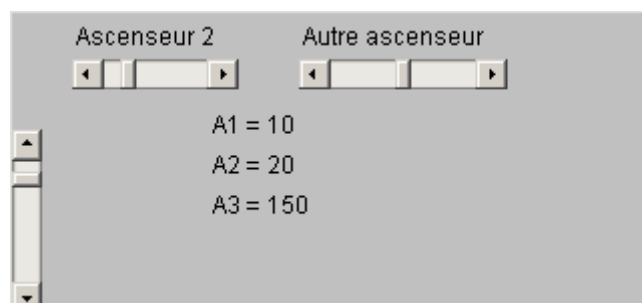
import java.applet.*;
import java.awt.*;

public class ascenseur extends Applet
{
    int v1=10,v2=20,v3=150; //valeurs initiales
    Font font = new Font("Helvetica",0,12);
    Panel panN = new Panel();
    Panel pan0=new Panel(),pan1 = new Panel();
    BorderLayout bl= new BorderLayout(20,20);
    Scrollbar sc1=new Scrollbar(1,v1,5,0,105); //vertical
    Scrollbar sc2=new Scrollbar(0,v2,10,0,110); //horizontal
    Scrollbar sc3=new Scrollbar(0,v3,10,100,210);
    Label lb2=new Label("Ascenseur 2");
    Label lb3=new Label("Autre ascenseur");
    public void init( )
    {
        setBackground(Color.lightGray);
        setFont(font);
        setLayout(bl);
        add("North",panN);    add("West",sc1);
        panN.setLayout(new FlowLayout(0,30,0)); //panneau Nord global
        panN.add(pan0); panN.add(pan1); //sous-panneaux Nord
        pan0.setLayout(new BorderLayout(5,0)); //pour placer lb2 et sc2
        pan0.add("North",lb2); //la longueur du Label détermine celle
        pan0.add("South",sc2); //de l'ascenseur
        pan1.setLayout(new BorderLayout(5,0)); //pour lb3 et sc3
        pan1.add("North",lb3);
        pan1.add("South",sc3);}

    public boolean handleEvent(Event evt)
    {
        if (evt.target==sc1) v1=sc1.getValue();
        else if (evt.target==sc2) v2=sc2.getValue();
        else if (evt.target==sc3) v3=sc3.getValue();
        else return super.handleEvent(evt);
        repaint();    return true;}

    public void paint(Graphics g)
    {
        g.drawString("A1 = "+v1,100,60);
        g.drawString("A2 = "+v2,100,80);
        g.drawString("A3 = "+v3,100,100);}
}

```



11.7 Autres composants

On peut aussi considérer les panneaux (**Panel**) et les canevas (**Canvas**) comme des composants : ils sont également mis en place par les protocoles de mises en page. La classe `Panel` a déjà été examinée lors de l'étude des gestionnaires de mise en page. Les canevas sont des zones de dessins pour lesquelles il faut obligatoirement redéfinir une méthode `paint(Graphics g)` spécifique. Il faut donc définir une classe propre à chaque canevas que l'on veut mettre en place. Le constructeur de cette classe est vide mais la méthode `resize()` permet de préciser ses dimensions.

12. Les composants graphiques (1.1)

A partir de la version 1.1, chaque composant a été doté de son propre détecteur d'événements (Listener). Le principe de détection des événements est le même que pour la souris et le clavier. Il faut importer le package **java.awt.event** puis pour chaque composant créer une méthode spécifique de traitement par l'ajout dans l'en-tête de l'applet de *implements xxxListener*, dans `init()` de *nom_composant.addxxxListener(this)* et d'implémenter les méthodes de traitement des événements. S'il faut utiliser plusieurs interfaces, les clauses qui suivent "implements" doivent être séparées par des virgules.

Pour **tous** les composants, il est possible de préciser, lors de la création, la fonte associée au libellé du composant [méthode `setFont()`] et la couleur du pinceau du libellé au moyen de la méthode `setForeground()`. Il est possible de cacher un composant de nom *cp* par la méthode `cp.setVisible(false)`, de le réafficher avec `cp.setVisible(true)`, de le désactiver avec `cp.setEnabled(false)` et enfin de le réactiver avec `cp.setEnabled(true)`.

Ces méthodes remplacent les méthodes `hide()`, `show()`, `disable()` et `enable()` de la version 1.0.

Les méthodes de créations des composants et de manipulations des informations sont identiques à celles de la version 1.0

12.1 Boutons et Zones de texte

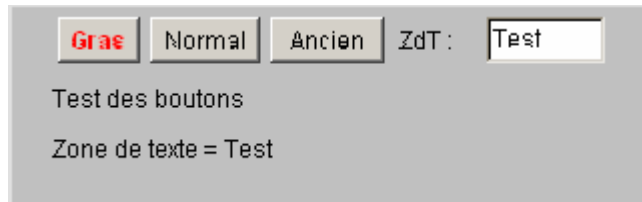
Ces composants répondent à un événement de type **action** (click souris pour les boutons, touche [Entrée] pour les zones de texte). Pour traiter les événements, il faut mettre en place l'interface **ActionListener**, doter chaque composant *cpx* d'un "écouteur" avec `cpx.addActionListener(this)` et enfin implémenter la méthode de traitement de l'événement `actionPerformed(ActionEvent evt)` qui retourne les objets "ActionEvent". La nature du composant générateur de l'événement peut être récupéré avec la méthode `evt.getSource()`. L'exemple ci-dessous reprend celui du chapitre 11. Les principales modifications sont indiquées en rouge dans le listing.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class bouton11 extends Applet implements ActionListener
{
    Font font = new Font("Helvetica",0,12);
    Font bold = new Font("Helvetica",1,12);
    boolean gras;    String s="Test";
    Label lb = new Label("ZdT :");
    Button bt1, bt2, bt3;
    TextField tf1 = new TextField(s,5);
    public void init()
    {
        setBackground(Color.lightGray); //gestionnaire par défaut
        setFont(font);
        bt1 = new Button("Gras"); //création
        bt1.setForeground(Color.red); //couleur du label en rouge
        bt1.setFont(bold); //fonte du label
        add(bt1);    bt1.addActionListener(this); //this => applet
        bt2 = new Button("Normal");
        add(bt2);    bt2.addActionListener(this);
        bt3 = new Button("Ancien");
        add(bt3);    bt3.addActionListener(this);
        add(lb);
        add(tf1);    tf1.addActionListener(this);
    }
}
```

```

    public void actionPerformed(ActionEvent evt)//remplace la méthode
action
    {
        if (evt.getSource().equals(bt1)) gras = true;
        else if (evt.getSource().equals(bt2)) gras = false;
        else if (evt.getSource()==bt3){
            if (bt3.getLabel()==" Ancien "){
                bt3.setLabel("Nouveau");
                bt1.setVisible(false);
                bt2.setEnabled(false);}
            else {
                bt3.setLabel(" Ancien ");
                bt1.setVisible(true);
                bt2.setEnabled(true);}}
        else if (evt.getSource()==tf1) s=tf1.getText();
        repaint();}
    public void paint(Graphics g)
    {
        if (gras) g.setFont(bold); else g.setFont(font);
        g.drawString("Test des boutons",20,50);
        g.drawString("Zone de texte = "+s,20,75);}
}

```



12.2 Listes et cases à cocher

Ces composants répondent aux événements de type sélection d'un item d'une liste ou à un click sur une case. Pour traiter les événements, il faut mettre en place l'interface **ItemListener**, doter chaque composant *cpx* d'un "écouteur" avec **cpx.addItemListener(this)** et enfin implémenter la méthode de traitement de l'événement **itemStateChanged(ActionEvent evt)** qui retourne les objets "ItemEvent". Ici encore la nature du composant générateur de l'événement peut être récupéré par la méthode **evt.getSource()**. Contrairement à la version 1.0, les listes répondent aux clicks simples et doubles sur un item de la liste.

L'exemple ci-dessous reprend celui du chapitre 11. Les principale modification sont indiquées en rouge dans le listing.

```

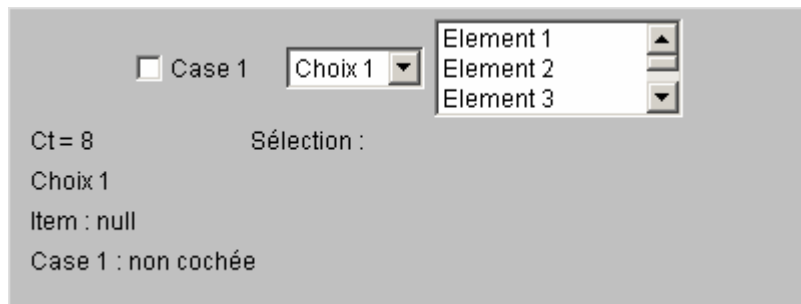
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class listell extends Applet implements ItemListener
{
    Font font = new Font("Helvetica",0,12);
    Choice choix;
    List liste;
    Checkbox cbl=new Checkbox("Case 1",false);//false => case non cochée
    int nch,nliste,ct;
    boolean ok;
    String s,s1,sel[]=new String[5];//tableau de chaînes
    public void init()
    {
        setBackground(Color.lightGray);
        setFont(font);
        add(cbl);          cbl.addItemListener(this); //création de l'écouteur
        choix = new Choice();
        choix.addItem("Choix 1");          choix.addItem("Choix 2");
        choix.addItem("Choix 3");
    }
}

```

```

add(choix); choix.addItemListener(this);
liste = new List(3,true); //3 lignes affichées, choix multiples
liste.addItem("Element 1");    liste.addItem("Element 2");
liste.addItem("Element 3");    liste.addItem("Element 4");
liste.addItem("Element 5");
add(liste); liste.addItemListener(this);}
public void itemStateChanged(ItemEvent evt)
{ if (evt.getSource()==choix){
    nch=choix.getSelectedIndex();
    s=choix.getSelectedItem();
    if (nch==2) ct=0;}
else if (evt.getSource().equals(liste))
    sel=liste.getSelectedItems(); //noter le s final
else if (evt.getSource().equals(cb1)) ok=!ok;
repaint();}
public void paint(Graphics g)
{ ct++;
  g.drawString("Ct = "+ct,10,70);
  g.drawString("Choix "+(nch+1),10,90);
  g.drawString("Item : "+s,10,110);
  g.drawString("Sélection : ",120,70);
  for (int i=0; i<sel.length; i++)
    g.drawString(sel[i],150,85+15*i);
  s1 = (ok) ? "cochée" : "non cochée";
  g.drawString("Case 1 : "+s1,10,130);}
}

```



12.3 Ascenseurs

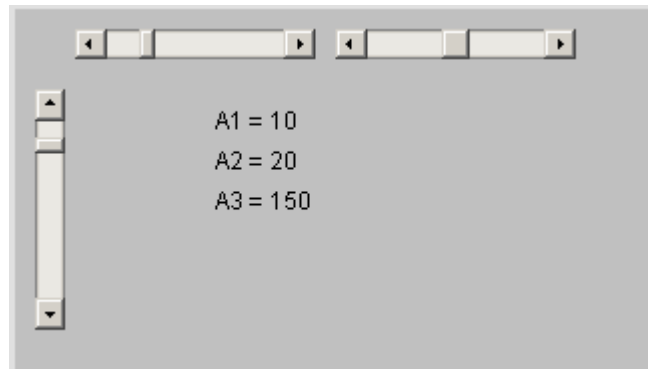
Les barres de défilement relèvent de l'interface **AdjustmentListener**. La prise en compte des événements de la barre *scx* est réalisée par la mise en place de la méthode **scx.addAdjustmentListener(this)**. La méthode **adjustmentValueChanged(AdjustmentEvent evt)** est invoquée à chaque modification de la barre de défilement. L'identité du composant générateur de l'événement peut être récupéré par la méthode **evt.getSource()**. La position du curseur est connue avec la méthode **getValue()**.

Dans l'exemple suivant, les ascenseurs sont placés sans utiliser de protocole de mise en page avec la méthode **setBounds(int xi, int yi, int large, int haut)**. La comparaison avec l'exemple du chapitre 11 qui utilise des panneaux imbriqués montre tout l'intérêt de cette méthode.

```

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class ascent11 extends Applet implements AdjustmentListener
{ int v1=10,v2=20,v3=150; //valeurs initiales
  Font font = new Font("Helvetica",0,12);
  FlowLayout fl = null; //pas de protocole
  Scrollbar sc1=new Scrollbar(1,v1,5,0,105);
  Scrollbar sc2=new Scrollbar(0,v2,10,0,110);
  Scrollbar sc3=new Scrollbar(0,v3,20,100,220);
public void init()
{ setBackground(Color.lightGray);
  setFont(font);
  setLayout(fl);
  add(sc1);      sc1.setBounds(10,40,15,120);//mise en place
  sc1.addAdjustmentListener(this); //écouteur
  add(sc2);      sc2.setBounds(30,10,120,15);
  sc2.addAdjustmentListener(this);
  add(sc3);      sc3.setBounds(160,10,120,15);
  sc3.addAdjustmentListener(this);}
public void adjustmentValueChanged(AdjustmentEvent evt)
{ if (evt.getSource()==sc1)
  v1=sc1.getValue();
  else if (evt.getSource()==sc2)
  v2=sc2.getValue();
  else if (evt.getSource()==sc3)
  v3=sc3.getValue();
  repaint();}
public void paint(Graphics g)
{ g.drawString("A1 = "+v1,100,60);
  g.drawString("A2 = "+v2,100,80);
  g.drawString("A3 = "+v3,100,100);}
}

```



Remarque : La manipulation intensive d'un ascenseur quand le navigateur est Netscape 4.x conduit souvent à des affichages incohérents de cet ascenseur.

13. Les exceptions

Tous les langages modernes possèdent un mécanisme pour le traitement des erreurs qui peuvent se produire lors de l'exécution. Dans JAVA, en cas de fonctionnement anormal, une **exception** est déclenchée. C'est une instance de la classe Exception.

Si une méthode peut potentiellement générer une exception, il faut obligatoirement la traiter sans quoi il se produit une erreur lors de la compilation. Ainsi la méthode `run()` utilisée lors des animations peut provoquer l'exception "*InterruptedException*". Cette méthode doit toujours comporter les blocs `try` et `catch` qui sont utilisés dans JAVA pour la gestion des erreurs.

Par contre, le traitement des erreurs déclenchées par les classes de `java.lang` est facultatif car ces erreurs peuvent se produire à tout moment.

Une erreur non traitée dans un bloc de code se propage vers les blocs supérieurs.

13.1 Les exceptions

Pour introduire les exceptions, nous allons examiner un exemple simple. Dans l'applet suivante, l'utilisateur doit taper un nombre entier N et le programme effectue ensuite la division de 1000 par le nombre N.

```
import java.applet.*;
import java.awt.*;
public class excep0 extends Applet
{ int i=1000,resultat,n;
  String s="100";
  Font font = new Font("Helvetica",0,12);
  Label lb1,lb2;
  TextField tf1=new TextField(""+s,5);//entrée de N
  TextField tf2=new TextField("  ",5);//affichage résultat
public void init()
{ setBackground(Color.lightGray);
  setFont(font);
  lb1=new Label("N =",0);
  add(lb1);   add(tf1);//saisie du nombre N
  tf1.setForeground(Color.red);
  lb2=new Label("1000/N =",0);
  add(lb2);   add(tf2);
  tf2.disable();} //réservé à l'affichage

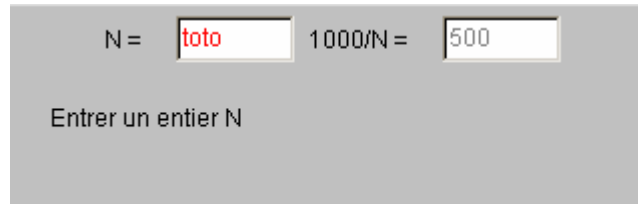
public boolean action(Event evt, Object arg)
{ if (evt.target==tf1) s=tf1.getText();
  else return super.action(evt,arg);
  repaint();   return true;}
public void paint(Graphics g)
{ g.drawString("Entrer un entier N",20,60);
  n=Integer.parseInt(s);//conversion chaîne vers entier
  resultat=i/n;
  tf2.setText(String.valueOf(resultat));}
}
```

Lors de l'exécution de ce programme, il peut se produire deux types d'erreurs.

a) l'utilisateur entre une chaîne qui correspond à un nombre réel ou une chaîne non convertible en nombre. Dans ces deux cas la machine virtuelle déclenche une exception "*NumberFormatException*" au niveau de l'instruction `n = Integer.parseInt(s)`.

b) l'utilisateur entre un zéro. La machine virtuelle déclenche alors une exception "*ArithmeticException*" lors de l'exécution de $resultat = i/n$.

Au niveau de l'applet apparemment rien ne se passe (tf2 reste inchangé) mais si l'on **examine la console JAVA**, on constate que la machine virtuelle émet des messages qui traduisent la détection puis la remonté de l'erreur puisqu'elle n'est pas traitée. On peut, au passage, noter le nombre de couches logicielles mises en oeuvre (et ce de manière transparente pour le programmeur) par la machine virtuelle.



N = 1000/N =

Entrer un entier N

Pour N, taper par exemple "2.5" puis "toto" et enfin "0"

13.2 Capture et traitement des exceptions

Pour capturer l'exception, il suffit d'utiliser les instructions **try** (essayer) et **catch** (attraper). Dans le bloc *try*, on place toutes les instructions qui peuvent produire une erreur. L'instruction *catch* précise la nature de l'exception qui peut survenir. Cette instruction **doit toujours être suivie d'un bloc** (délimité par deux accolades { et }) même si ce bloc est vide ou contient une seule instruction. Dans le bloc, on peut traiter ou non l'exception.

Si plusieurs exceptions peuvent se produire dans le bloc *try*, on peut utiliser successivement plusieurs clauses *catch*.

On peut modifier l'exemple ci-dessus en remplaçant sa méthode *paint()* par la méthode ci-dessous qui traite les deux exceptions possibles. Dans les variables de classe de l'applet, il faut ajouter les booléens *err1* et *err2*. Le bloc *try* contient les deux instructions pouvant provoquer une erreur.

En cas de division par 0, le bloc qui suit l'instruction *catch(ArithmeticException err)* positionne le booléen *err1* à *false*. Il est possible de récupérer avec *err* un message sur la nature précise de l'erreur.

Si *s* ne représente pas un entier, le bloc qui suit l'instruction *catch(NumberFormatException e)* positionne le booléen *err2* à *false*.

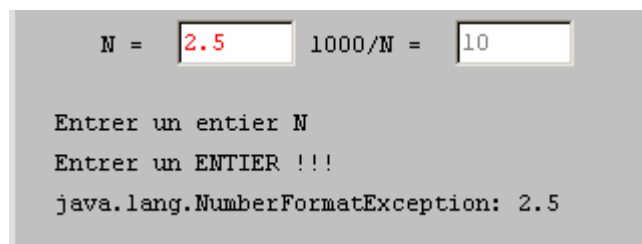
La suite de la méthode affiche selon les cas un message d'erreur ou le résultat final. Il faut bien sûr repositionner les booléens pour pouvoir continuer à utiliser le programme avec des données valides.

Si l'on introduit dans le code la clause *try* et la clause *catch* qui correspond à l'erreur, la machine virtuelle n'émet plus de messages même si le bloc *catch* est vide.

```

public void paint(Graphics g)
{ g.drawString("Entrer un entier N",20,60);
  try{ //bloc try
    n=Integer.parseInt(s);
    resultat=i/n;}//fin du bloc try
  catch (ArithmeticException e) {err1=true;}// { et } impératifs
  catch (NumberFormatException e) {err2=true;}
  if (err1){
    g.drawString("Division par 0",20,80);
    g.drawString(""+e,20,100);
    err1=false;} //pour la saisie suivante !
  else if (err2){
    g.drawString("Entrer un ENTIER !!!",20,80);
    g.drawString(""+e,20,100);
    err2=false;}
  else tf2.setText(String.valueOf(resultat));}

```



Pour N, taper par exemple "2.5" puis "toto" et enfin "0"

Examiner la console JAVA et vérifier que la machine virtuelle n'envoie plus de messages quand on tape une valeur de N non valide.

13.3 Clause finally

Quand on désire que certaines instructions soient exécutées même en cas d'erreur, on peut inclure à la suite des clauses *catch* un bloc *finally*. Si l'exception détectée correspond à l'un des blocs *catch*, les instructions de ce bloc sont exécutés et ensuite on exécute celle du bloc *finally*. Si aucun *catch* ne correspond à l'exception générée, le bloc *finally* est immédiatement exécuté. La méthode *paint* ci-dessous utilise cette technique.

```

public void paint(Graphics g)
{ err1=false; err2=false;//effacement de erreurs antérieures
  g.drawString("Entrer un entier N",20,60);
  try{
    n=Integer.parseInt(s);
    resultat=i/n;}
  catch (ArithmeticException e) {
    err1=true;
    g.drawString("Division par 0",20,80);}
  catch (NumberFormatException e) {
    err2=true;
    g.drawString("Entrer un ENTIER !!!",20,80);}
  finally{ //toujours exécuté
    if (!err1 && !err2)
      tf2.setText(String.valueOf(resultat));}}

```

13.4 Liste des exceptions

La majorité des packages de JAVA peuvent générer des exceptions. JAVA effectue à la compilation et à l'exécution de multiples contrôles : c'est pourquoi on peut considérer que c'est un langage sûr.

Pour obtenir la liste complète des exceptions, il faut examiner dans la documentation JAVA la liste des classes de chaque package.

Pour le package java.lang les exceptions les plus souvent rencontrées sont ArithmeticException, NumberFormatException, IllegalArgumentException (par exemple racine carrée d'un nombre négatif), ArrayIndexOutOfBoundsException (indice de tableau incorrect), NullPointerException (tentative d'utilisation d'un objet déclaré mais non créée) et InterruptedException qui doit obligatoirement être traitée.

14. Les classes de java.util

Le package java.util est un package "fourre-tout" qui contient des classes utiles pour la résolution de certains problèmes.

14.1 Classe Date

Le constructeur de cette classe peut prendre les formes Date(), Date(long date), Date(int year, int month, int date), Date(int year, int month, int date, int hrs, int min), Date(int year, int month, int date, int hrs, int min, int sec) et Date(String s). Dans ce dernier cas, la chaîne est de la forme "day, date month year hrs:min:sec GMT" dans laquelle day est le nom du jour sous la forme Mon, Tue, Wed, Thu, Fri, Sat ou Sun. Dans les versions 1.0.x, les dates démarrent le 1 janvier 1970.

La version Date() correspond à la date courante. Dans la version Date(long date), date correspond au nombre de millisecondes écoulées entre le 1-01-1970 minuit et la date à définir. Les méthodes les plus utiles sont getTime() qui retourne un long correspond au nombre de millisecondes écoulées entre le 1-01-1970 et la date utilisée, getDate() qui retourne un int correspondant à la date du jour, getDay() qui retourne le jour, getMonth(), getYear(), getHours(), getMinutes().

Les méthodes set correspondantes permettent de fixer les mêmes paramètres d'une date donnée.

Une originalité : les mois vont de 0 (janvier) à 11 (décembre).

L'exemple ci-dessous met en oeuvre un certain nombre de ces méthodes. On demande à l'utilisateur d'entrer sa date de naissance et on retourne son âge.

```

import java.applet.*;
import java.awt.*;
import java.util.*; //importation du package
public class date10 extends Applet
{ int jour=15,mois=8,annee=70; //valeurs initiales
  Font font = new Font("Helvetica",0,12);
  Label lbl1,lb2,lb3;
  TextField tf1=new TextField(""+jour,2);
  TextField tf2=new TextField(""+(mois-1),2);
  TextField tf3=new TextField(""+annee,2);
  Date D = new Date(); //date actuelle du client
  String s;
public void init()
{ setBackground(Color.lightGray);
  setFont(font);
  lbl1=new Label("Jour",2);
  add(lbl1);  add(tf1);
  lb2=new Label("Mois",2);
  add(lbl2);  add(tf2);
  lb3=new Label("Année",2);
  add(lbl3);  add(tf3);}
public boolean action(Event evt, Object arg)
{ if (evt.target==tf1) {
  s=tf1.getText();  jour=Integer.parseInt(s);}
  else if (evt.target==tf2) {
  s=tf2.getText();  mois=Integer.parseInt(s)-1;}//0 à 11 !!!
  else if (evt.target==tf3){
  s=tf3.getText();  annee=Integer.parseInt(s);}
  else return super.action(evt,arg);
  repaint();  return true;}
public void paint(Graphics g)
{ g.drawString("Nous sommes le : "+D,20,50);
  g.drawString("Entrer votre date de naissance :",20,70);
  Date Dnai = new Date(annee,mois,jour);
  g.drawString("né(e) le : "+Dnai.toString(),20,90);
  Date Dage = new Date((long)(D.getTime()-Dnai.getTime()));
  s = "Vous avez "+(Dage.getYear()-70)+" ans, "; //origine 1970
  s = s+Dage.getMonth()+" mois et ";
  s = s+(Dage.getDate()-1)+" jours.";
  g.drawString(s,20,110);
  g.drawString(""+D.getTime()+" ms écoulées depuis le 01/01/1970
minuit",20,130);
}

```

Jour Mois Année

Nous sommes le : Sun Nov 07 16:07:10 GMT+01:00 2004
 Entrer votre date de naissance :
 né(e) le : Tue Sep 15 00:00:00 GMT+02:00 1970
 Vous avez 34 ans, 1 mois et 23 jours.
 1099840030781 ms écoulées depuis le 01/01/1970 minuit

Valider chaque saisie.

14.2 Classe StringTokenizer

Cette classe permet de découper une chaîne de caractères en fonction de séparateurs (virgule, espace ...). Les utilisateurs des anciens BASIC retrouvent avec cette classe l'équivalent des instructions DATA.

Le constructeur *StringTokenizer(String s, String sep)* comporte deux paramètres : la chaîne *s* à découper et *sep* qui est une chaîne contenant les séparateurs à utiliser pour le découpage en éléments.

La méthode *hasMoreTokens()* retourne true tant que la chaîne *s* contient des éléments à découper. La méthode *nextToken()* retourne une chaîne qui contient l'élément en question. On utilise en général (comme dans l'exemple ci-dessous) une boucle while pour décomposer la chaîne *s* en ses différents éléments.

```
import java.applet.*;
import java.awt.*;
import java.util.*; //importer le package
public class version extends Applet
{   String s="10,15,24,36,,78;47,26 38,45,9"//chaîne à découper
    int Tab[] = new int[10]; //tableau des valeurs
    public void paint(Graphics g)
    {   int i=0;
        StringTokenizer st =new StringTokenizer(s,"; ,"); //3 séparateurs ; , et
        blanc
        while (st.hasMoreTokens()){ //boucle de lecture
            Tab[i]=Integer.parseInt(st.nextToken()); //éléments du tableau
            g.drawString(""+Tab[i],20,20+15*i);
            i++;}}
    }
```

Remarques

a) dans l'exemple ci-dessus, il serait bien plus simple de procéder directement à l'affectation du tableau au moyen de l'instruction :

```
Tab[] = {10,15,24,36,78,47,26,38,45,9};
```

b) noter que la seconde virgule entre 36 et 78 sera ignorée.

14.3 Classe Random

Cette classe permet de générer des nombres pseudo-aléatoires de façon plus complète que la méthode *Math.random()* de *java.lang* qui à chaque appel renvoie un double compris entre 0.0d et 1.0d .

Après appel du constructeur *Random rnd = new Random()* ; , il suffit d'utiliser la méthode qui correspond au type de nombre aléatoire que l'on désire obtenir.

rnd.nextInt() retourne un entier int compris entre *Integer.MIN_VALUE* (-2147488348) et *Integer.MAX_VALUE* (2147488347).

rnd.nextLong() retourne un entier long compris entre *Long.MIN_VALUE* et *Long.MAX_VALUE*.

rnd.nextDouble() et *rnd.nextFloat()* retournent un double ou un float compris entre 0 et 1.

rnd.nextGaussian() retourne un double avec une distribution gaussienne de moyenne 0.0 et de déviation 1.0.

Enfin la méthode *setSeed(long seed)* permet de fixer la valeur de la "graine" utilisée par le générateur pseudo-aléatoire. En fixant la valeur de la graine, on obtient toujours la même séquence de valeurs pseudo-aléatoires lors d'exécutions successives du programme.

14.4 Classe Vector

La classe `Vector` permet de créer et de manipuler des **tableaux d'objets dynamiques**. La dimension du tableau peut en effet varier automatiquement quand on y insère de nouveaux éléments. Pour des raisons d'efficacité, il est conseillé d'accroître sa taille avant l'ajout d'un nombre important d'éléments afin de limiter les réallocations mémoire au minimum. Il n'y a pas de restrictions sur la nature des objets (on peut même mélanger des objets différents dans un `Vector`).

Les constructeurs sont `Vector()` qui crée un tableau vide de dimension nulle, `Vector(int nb)` qui crée un tableau vide de dimension `nb` et `Vector(int nb, int inc)` qui crée un tableau vide de dimension `nb` et dont la taille augmente de `inc` à chaque fois que la taille maximum est atteinte.

Il n'est pas possible de stocker de nombres dans un vecteur mais il est possible de stocker des instances des classes `Integer`, `Double` ... car ce sont des objets.

Les principales méthodes de cette classe sont :

`addElement(Object obj)` ajoute un élément à la fin de la liste.

`capacity(int nb)` retourne la taille actuelle.

`contains(Object obj)` retourne un booléen indiquant la présence ou l'absence de l'objet `obj` dans le `Vector`.

`elementAt(int index)` retourne l'élément situé à la place `index`.

`indexOf(Object obj)` donne l'index de l'objet `obj`.

`insertElementAt(Object obj, int index)` insère l'objet `obj` à la place `index`. Les éléments suivants l'index sont décalés vers le bas.

`setElementAt(Object obj, int index)` modifie l'objet à la place `index`.

`removeElementAt(int index)` supprime l'objet situé à `index`. Les éléments suivants l'index sont décalés vers le haut.

`setSize(int nb)` fixe la taille actuelle.

De la classe `Vector` dérive la **classe Stack** qui implémente une **pile d'objets**. La méthode `push()` place un objet sur le haut de la pile, `pop()` supprime l'objet situé sur le haut de la pile et la méthode `peek()` retourne l'objet situé sur le haut de la pile sans l'enlever.

14.5 Classe Hashtable

Cette classe implémente une structure qui à chaque donnée associe une clé. C'est un **tableau d'objets** qui stocke ceux-ci avec des indices non numériques.

Les constructeurs sont `Hashtable()` qui crée une table vide de dimension nulle, `Hashtable(int nb)` qui crée une table vide de dimension `nb` et `Hashtable(int nb, float inc)` qui crée une table vide de dimension `nb` et dont la taille augmente quand le nombre d'éléments de la table dépasse le produit `nb*inc`. (`inc` doit être compris entre 0.0f et 1.0f).

Les principales méthodes de cette classe sont :

`put(Object key, Object data)` ajoute un élément et sa clé à la table.

`get(Object key)` retourne l'élément de la table dont la clé est `key`.

`remove(Object key)` supprime l'élément de la table dont la clé est `key`.

15. Un exemple complet

15.1 Analyse du projet

On se propose de réaliser une applet qui permet de visualiser la gestion des couleurs utilisables dans les modes RVB (rouge, vert, bleu) et HSB (teinte, saturation, luminosité). Pour modifier la couleur dans chaque mode, il faut introduire trois nombre entiers. En mode RVB chaque valeur est comprise entre 0 et 255. En mode HSB la couleur est un nombre compris entre 0 et 360, la saturation et la luminosité sont compris entre 0 et 100. On fera un contrôle de validité des valeurs entrées. On se propose d'afficher la couleur obtenue dans un cadre et d'utiliser la couleur complémentaire comme couleur du fond d'une partie de l'applet. Le programme proposé offre une solution possible parmi beaucoup d'autres. Il existe en particulier de nombreuses possibilités pour effectuer la mise en place des composants nécessaires au fonctionnement.

Pour le cadre, on fait le choix d'utiliser un canevas (**Canvas**). Ce composant impose la surcharge de sa méthode `paint()` : il faut donc définir obligatoirement une classe spécifique. Nous utilisons la classe *Cadre* qui étend la classe *Canvas*. Cette classe contient uniquement la variable publique *Color col* qui sera utilisée comme couleur de pinceau et la méthode `paint()`.

La saisie des valeurs de la couleur en mode RVB ou HSB suppose la mise en place pour chaque cas de 3 zones de texte. Au lieu de définir 6 zones, il est plus simple de définir deux objets identiques qui seront paramétrés pour chacun des deux cas. On va donc construire une classe *Panneau* qui va étendre la classe *Panel*. Cette classe va comporter un constructeur permettant le paramétrage des zones de texte et des libellés associés et la méthode `action()` qui va implémenter la réaction du programme aux commandes de l'utilisateur. Pour chaque panneau, il faut introduire 6 composants (3 labels et 3 zones de texte) L'utilisation d'un gestionnaire du type *GridLayout* avec 2 colonnes de 3 lignes semble être le meilleur choix. La classe principale est l'applet. La méthode `init()` effectue la mise en place des deux panneaux et du canevas avec un gestionnaire *FlowLayout*. Un gestionnaire *GridLayout* avec 1 ligne de 3 colonnes est aussi utilisable. La réponse aux actions de l'utilisateur est implémentée dans la méthode `mise_a_jour` mais le code correspondant à cette méthode aurait pu être introduit dans la méthode `action` de la classe *Panneau*. La méthode `paint()` de cette classe assure la mise à jour finale de l'affichage.

15.2 Listing commenté

```
import java.applet.*;
import java.awt.*;
public class ColorTest extends Applet
{ Panneau panRGB,panHSB; //déclaration des objets Panneau
  Cadre canv; //déclaration objet Cadre
  Color c1,c2;
  Font font = new Font("Helvetica",0,11);
  Font bold = new Font("Helvetica",1,18);
public void init()
{ setFont(font);//la taille de la fonte conditionne celle des panneaux
  setBackground(Color.lightGray); //couleur du fond de l'applet
  setLayout(new FlowLayout(1,40,5)); //protocole de mise en page global
  canv = new Cadre(); //création du Cadre
  panRGB = new Panneau(this,"Rouge","Vert","Bleu"); //appel constructeur
  panHSB = new Panneau(this,"Teinte","Saturation","Luminosité");
  add(panRGB); //ajout ler panneau
  add(canv); //ajout du Cadre à coté du panneau
  canv.setFont(bold);//fonte du Cadre
  canv.resize(100,60); //taille du Cadre
  add(panHSB);
  panRGB.tf2.setText("255");//initialise une zone de texte à une valeur
non nulle
  mise_a_jour(panRGB);} //initialisation globale avec le vert maximum
public void paint(Graphics g)
{ g.setFont(bold); //fonte pour l'applet
  g.setColor(c2); //couleur du pinceau
  g.fillRect(50,100,350,80);
  g.setColor(c1); //couleur d'écriture
  g.drawString("Dans l'applet",160,150);
  g.setColor(Color.black);
  g.drawRect(50,100,350,80);}
void mise_a_jour(Panneau pan) //argument = Panneau modifié
{ int v1 = Integer.parseInt(pan.tf1.getText());
/*conversion chaîne vers entier avec contrôle des valeurs
l'entrée d'une valeur non entière provoque une exception*/
  if (pan==panRGB){
    if ((v1<0) || (v1>255)){ //contrôle de validité 0-255
      v1=128; pan.tf1.setText("128");}}
    else
      if ((v1<0) || (v1>360)){//contrôle de validité 0-360
        v1=0; pan.tf1.setText("0");}
  int v2 = Integer.parseInt(pan.tf2.getText());
  if (pan==panRGB){
    if ((v2<0) || (v2>255)){//contrôle de validité 0-255
      v2=128; pan.tf2.setText("128");}}
    else
      if ((v2<0) || (v2>100)){//contrôle de validité 0-100
        v2=50; pan.tf2.setText("50");}
  int v3 = Integer.parseInt(pan.tf3.getText());
  if (pan==panRGB){
    if ((v3<0) || (v3>255)){
      v3=128; pan.tf3.setText("128");}}
    else
      if ((v3<0) || (v3>100)){
        v3=50; pan.tf3.setText("50");}
  //Modification RVB
  if (pan==panRGB)
    { c1 = new Color(v1,v2,v3); //définition de la couleur en RVB
```

```

        float[] hsb = Color.RGBtoHSB(v1,v2,v3,(new float[3])); //conversion
en HSB
        hsb[0] *= 360; hsb[1] *= 100; hsb[2] *= 100; //conversion dans les
gammes choisies
        panHSB.tf1.setText(String.valueOf((int)hsb[0]));
        //mise à jour affichage dans les zones de texte du panneau RVB
        panHSB.tf2.setText(String.valueOf((int)hsb[1]));
        panHSB.tf3.setText(String.valueOf((int)hsb[2]));}
else //modification du panneau HSB
{   cl = Color.getHSBColor((float)v1/360,(float)v2/100,(float)v3/100);
    //conversion des valeurs entre 0.0f et 1.0f
    panRGB.tf1.setText(String.valueOf(cl.getRed()));
    //mise à jour affichage de l'autre panneau
    panRGB.tf2.setText(String.valueOf(cl.getGreen()));
    panRGB.tf3.setText(String.valueOf(cl.getBlue()));}
    int r=255-cl.getRed(); //couleur complémentaire
    int v=255-cl.getGreen();
    int b=255-cl.getBlue();
    c2=new Color(r,v,b);
    canv.setBackground(c1); canv.col=c2; //variables du Cadre
    canv.repaint(); //mise à jour du Cadre
    repaint(); //mise à jour de l'applet
}
//*****
class Panneau extends Panel
{   ColorTest applet; //variables de la classe panel
    TextField tf1,tf2,tf3;
Panneau(ColorTest app,String S1,String S2, String S3)//constructeur
{   applet = app;
    setLayout(new GridLayout(3,2,10,10)); //grille 2 colonnes de 3 lignes
    tf1 = new TextField("0"); //création des zones de texte
    tf2 = new TextField("0");
    tf3 = new TextField("0");
    add(new Label(S1,Label.LEFT));
        //création des labels avec les paramètres du constructeur
    add(tf1);
    add(new Label(S2,0));   add(tf2);
    add(new Label(S3,0));   add(tf3);}
public boolean action(Event evt, Object arg)
{   if (evt.target instanceof TextField){
        applet.mise_a_jour(this); //appel d'une méthode de l'applet
        return true;}
    else return false;}
}
//*****
class Cadre extends Canvas
{   Color col;
public void paint(Graphics g) //surcharge de la méthode
{   g.setColor(Color.black);
    g.drawRect(0,0,size().width-1,size().height-1);
    g.setColor(col);
    g.drawString("Canevas",10,20);}
}

```

15.3 L'applet

