Java langage orienté objet

Le contenu de ce thème :

Les classes

Java2

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

Les classes : des nouveaux types

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

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

Déclaration d'une classe

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

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

```
  procedure P2;
end;

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

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

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

Une classe est un type Java

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

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

Delphi

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

    Deux = class
      ...
    end;

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

    implementation

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

    end.
```

Java

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

  class Deux
  { ...
  }
```

Toutes les classes ont le même ancêtre - héritage

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

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

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

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

Une déclaration du type :

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

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

Comparaison héritage Delphi et Java :

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

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

Encapsulation des classes

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

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

Mots clefs pour la protection des classes et leur visibilité :

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

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

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

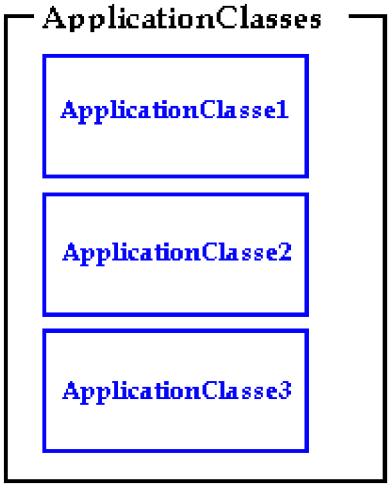
Nous étudions ci-après la visibilité des 3 classes précédentes dans deux contextes différents.

Exemple de classe intégrée dans une autre classe

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

classe publique.

Exemple correspondant à l'imbrication de bloc suivante :



La classe :

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

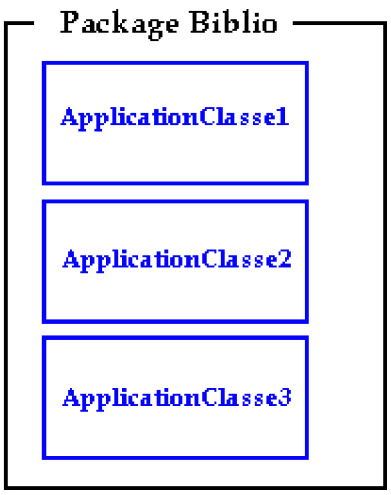
Un programme utilisant la classe :

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

Exemple de classe inclue dans un package

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

Exemple correspondant à l'imbrication de bloc suivante :



Le package :

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

Un programme utilisant le package :

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

Remarques pratiques :

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

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

import Pack1.Cl1;

Méthodes abstraites

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

Une méthode déclarée en abstract dans une classe mère :

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

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

Exemple de méthode abstraite :

```
class Etre_Vivant {
}
```

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

```
class Serpent extends Etre_Vivant {
}

class Oiseau extends Etre_Vivant {
}

class Homme extends Etre_Vivant {
}
```

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

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

class Serpent extends Etre_Vivant {
```

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

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

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

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

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

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

Classe abstraite

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

Vocabulaire et concepts :

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

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

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

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

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

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

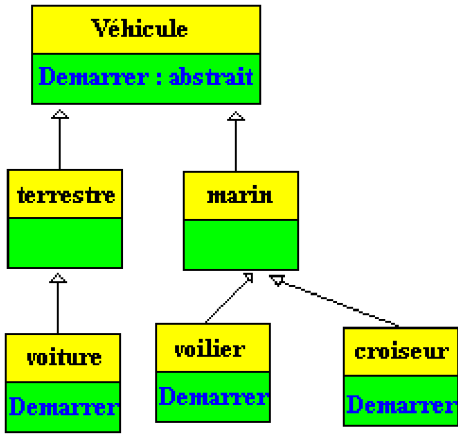
```
abstract class Etre_Vivant {
 abstract void SeDeplacer( );
}
```

Remarque

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

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

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



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

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

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

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

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

Dans cet exemple, supposons que :

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

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

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

Implantation d'un squelette Java de l'exemple

Java

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

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

Les objets

Java2

Les objets : des références

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

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

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

Modèle de la référence et machine Java

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

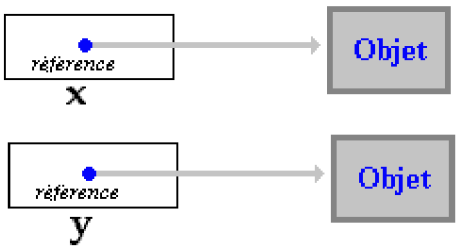
- La déclaration d'identificateur de variable typée qui contiendra la référence,
- la création de la structure de données elle-même (bloc objet de données) avec **new**.

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

```
    x , y : Un;
....

// la création :
x := Un.create ;
y := Un.create ;
```

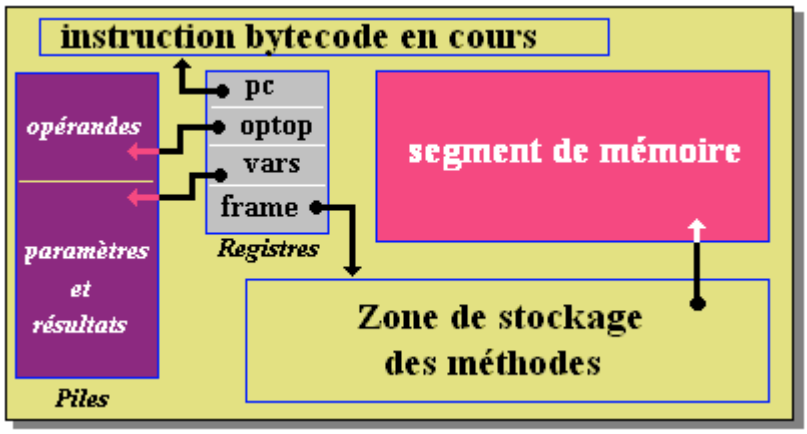
```
....

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

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

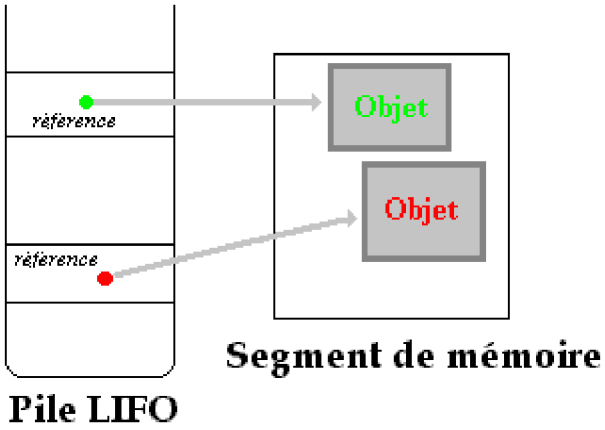


Un programme Java est fait pour être exécuté par une **machine virtuelle Java**, dont nous rappellons qu'elle contient 6 éléments principaux :

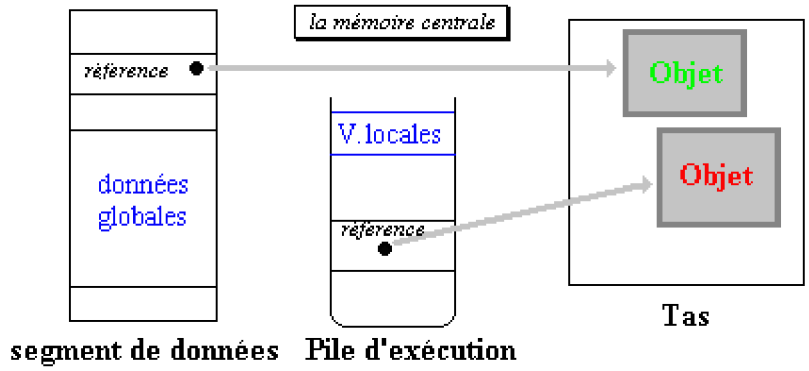
- Un jeu d'instructions en pseudo-code
- Une pile d'exécution LIFO utilisée pour stocker les paramètres des méthodes et les résultats des méthodes
- Une file FIFO d'opérandes pour stocker les paramètres et les résultats des instructions du p-code (calculs)
- Un segment de mémoire dans lequel s'effectue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



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

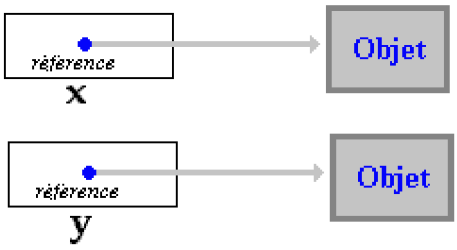


Attitude à rapprocher pour comparaison, à celle dont **Delphi** gère les objets dans une pile d'exécution de type LIFO et un tas :

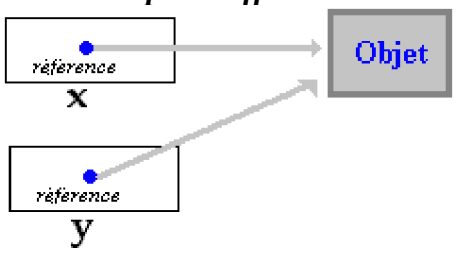


Attention à l'utilisation de l'affectation entre variables d'objets dans le modèle de représentation par référence. L'affectation x = y ne recopie pas le bloc objet de données de y dans celui de x, mais seulement la référence (l'adresse) de y dans la référence de x. Visualisons cette remarque importante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

Situation au départ, avant affectation



Situation après l'affectation " x = y "



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

Les constructeurs d'objets

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

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

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

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

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

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

Possibilités de définition des constructeurs :

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

Exemple de constructeur avec instructions :

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

Exemple de constructeur avec paramètre :

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

Exemple avec plusieurs constructeurs :

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

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

Comparaison Delphi - Java pour la déclaration de constructeurs

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

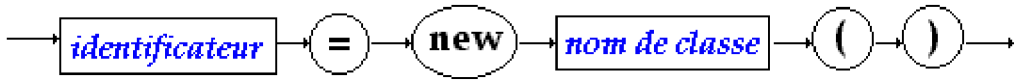
En Delphi un constructeur a un nom quelconque, tous les constructeurs peuvent avoir des noms différents ou le même nom comme en Java.

Utilisation du constructeur d'objet automatique (par défaut)

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

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

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



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

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

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

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

```
class Un
 { ...
 }

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

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

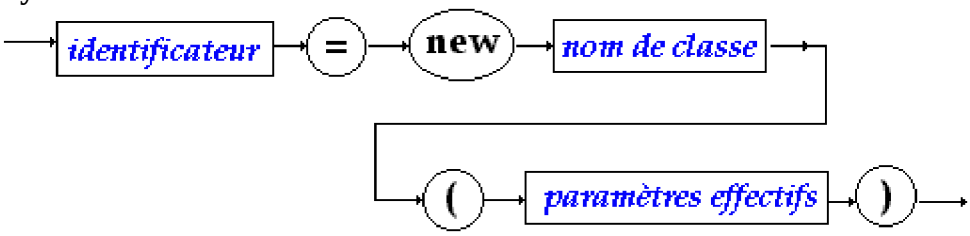
Java	Explication
class AppliClassesReferences **{** **public static void** main(String [] arg) **{** **Un** x,y ; x = **new Un**( ); y = **new Un**( ); System.out.println("x.a="+x.a); System.out.println("y.a="+y.a); y = x; x.a =12; System.out.println("x.a="+x.a); System.out.println("y.a="+y.a); **}**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

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

Utilisation d'un constructeur d'objet personnalisé

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

Syntaxe



Exemple avec plusieurs constructeurs :

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

}	

Le mot clef this pour désigner un autre constructeur

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

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

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

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

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

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

Comparaison Delphi - java sur cet exemple (similitude complète)

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

```
 end;

implementation

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

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

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

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

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

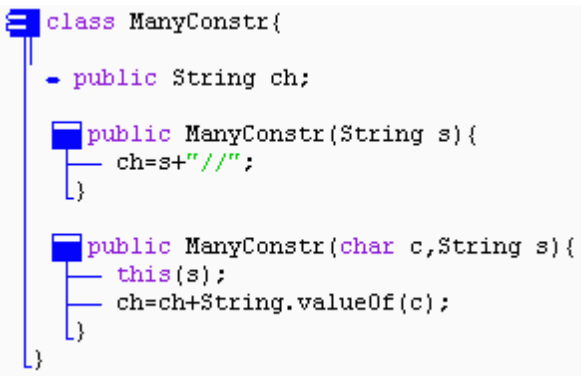
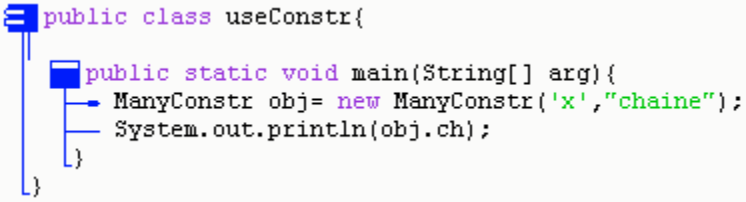
Delphi	Java
Un = **class** a : integer; **public** **procedure** methode(a:real); **end**; **implementation** **procedure Un**.methode(a:real);**begin** a = **self**.a + 7 ; **end**;	**class Un** **{** **int** a; **public void** methode(**float** a) **{** a = **this**.a + 7 ; **}** **}**

Le this peut servir à désigner une autre surcharge de constructeur

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

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

Exemple de classe à deux constructeurs :

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

Attributs et méthodes

Java2

Variables et méthodes

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

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

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

Variables dans une classe en général

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

Exemple de variables locales de méthode :

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

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

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

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

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

Exemple de attributs de classe :

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

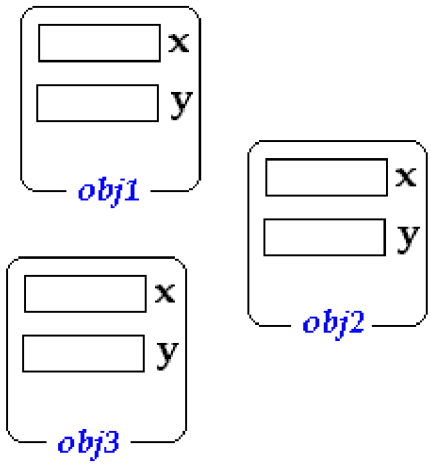
Les attributs de classe peuvent être soit de la catégorie des **variables de classe**, soit de la catégorie des **variables d'instance**.

Variables et méthodes d'instance

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

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

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



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

Programme Java exécutable

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

Variables et méthodes de classe - static

Variable de classe

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

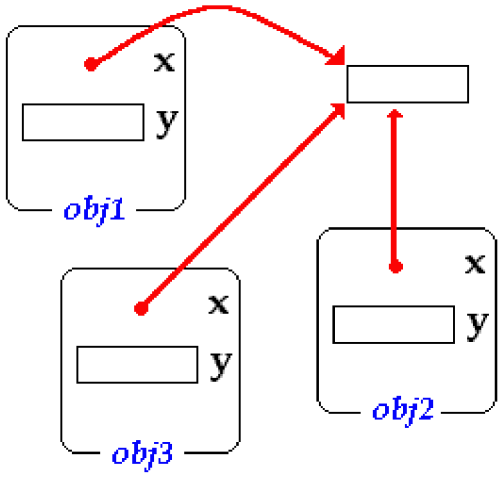
Voici deux déclarations de variables de classe :

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

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

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

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

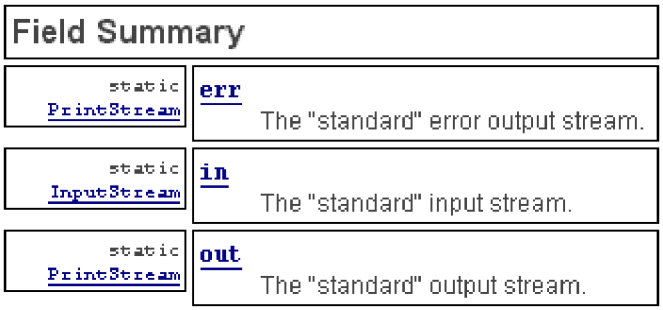


Exemple de variables de classe :

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

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

Les champs de la classe System :



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

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

Méthode de classe

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

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

Un programme correct illustrant le discours :

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Surcharge et polymorphisme

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

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

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

A - La surcharge de méthode (polymorphisme statique)

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

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

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

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

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

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

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

```
  { a *=10;  }

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

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

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

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

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

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

Comparaison Delphi - java sur la surcharge :

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

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

```
   }
}
```

B - La redéfinition de méthode (polymorphisme dynamique)

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

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

Comparaison redéfinition Delphi et Java :

Delphi

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

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

implementation

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

Java

```
class ClasseMere
{
  int x = 10;

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



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

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

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

Java

```
class ClasseMere
{
 int x = 10;

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

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

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

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

Soit à partir de l'exemple précédent les instructions suivantes :

```
ClasseFille obj = new ClasseFille( );
obj.g(-3,8);
obj.g('h');
```

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

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

Résumé pratique sur le polymorphisme en Java

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

Le mot clef super

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

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

Exemple :

```
class ClasseMere
{
  int x = 10;

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

class ClasseFille extends ClasseMere
{
```

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

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

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

Exemple :

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

}

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

Modification de visibilité

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

Modification de visibilité (modularité public-privé)

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

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

Ces attributs de visibilité sont identiques à ceux de Delphi.

Les interfaces

Java2

Introduction

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

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

Vocabulaire et concepts :

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

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

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

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

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

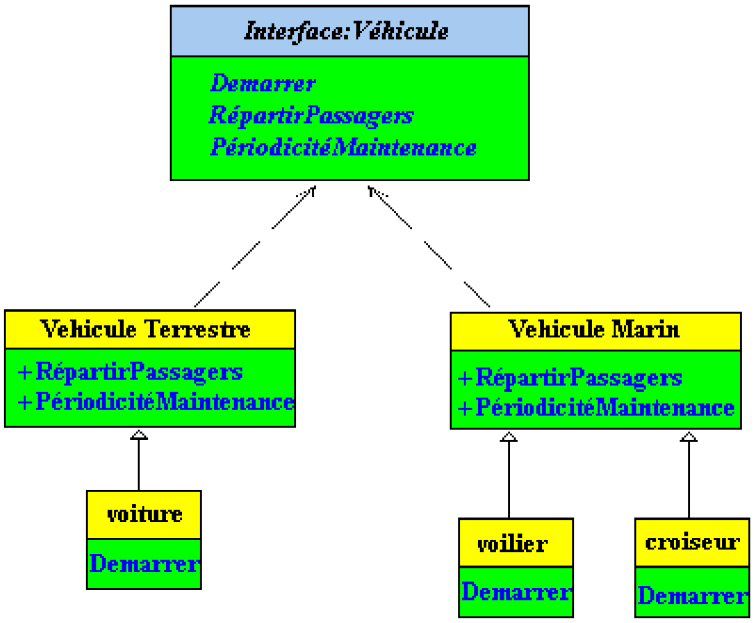
Si vous voulez utiliser la notion d'interface pour fournir un polymorphisme à une famille de classes, elles doivent toutes implémenter cette interface, comme dans l'exemple ci-dessous.

Exemple :

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

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

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



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

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

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

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

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

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

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

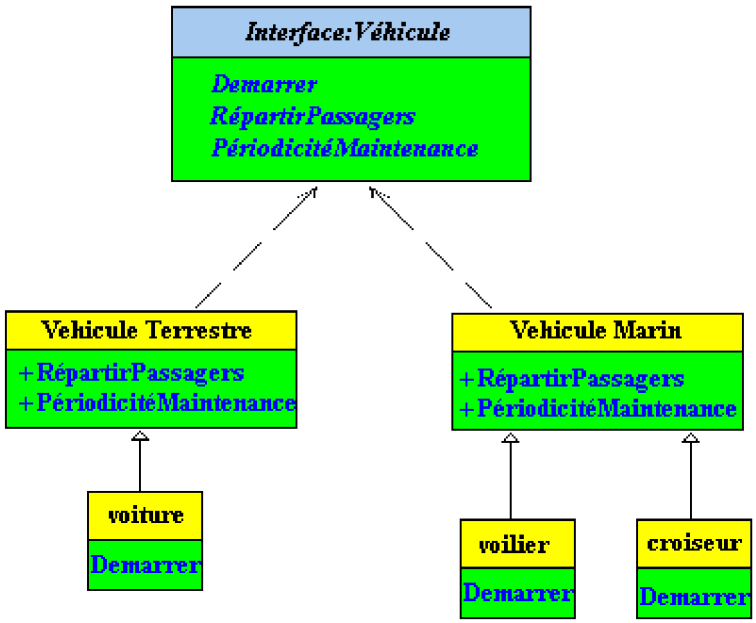
Utilisation pratique des interfaces

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

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

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

Exemple de hiérarchie à partir d'une interface :



Dans cet exemple :

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

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

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

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

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

abstract class Marin implements IVehicule {
```

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


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

Java2 à la fenêtre avec Awt

IHM avec Java

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

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

Le package AWT

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

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

Les classes Conteneurs

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

Hiérarchie de la classe* Container *:

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

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

Les principales classes conteneurs :

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

Les classes héritées des classes conteneurs :

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

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

Quelques méthodes de la classe Frame, utiles au départ :

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

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

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

Programme Java

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

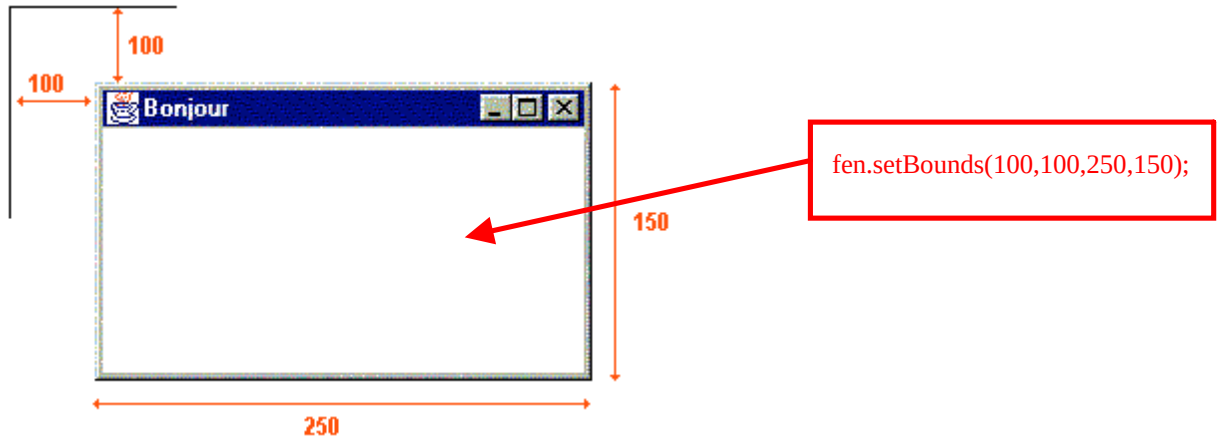
Ci-dessous la fenêtre affichée par le programme précédent :

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

Programme Java

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

Ci-dessous la fenêtre affichée par le programme précédent :



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

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

une fenêtre de classe Dialog :

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

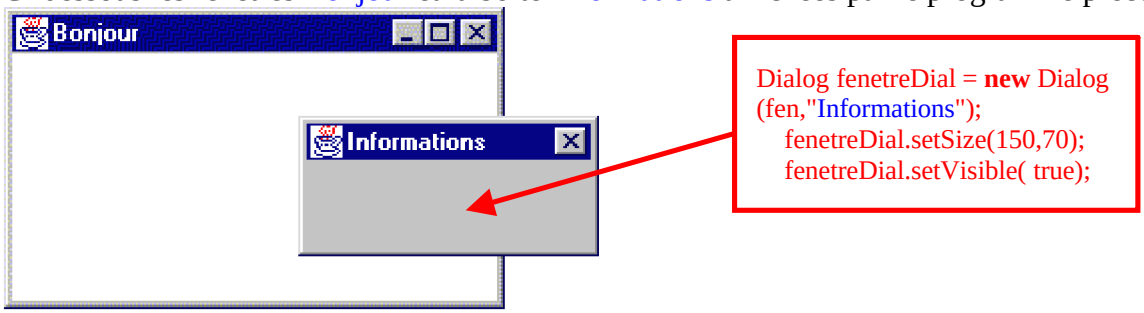
public Dialog(**Frame** owner, **String** title)
où owner est la Frame qui va appeler la boîte de dialogue, title est la string contenant le titre de la boîte de dialogue. Il faudra donc appeler le constructeur Dialog avec une Frame instanciée dans le programme.

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

Programme Java

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

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



Composants déclenchant des actions

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

Composants permettant le déclenchement d'actions :

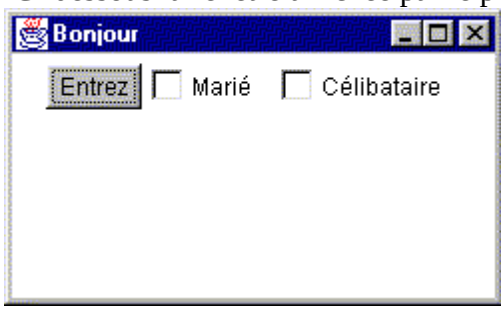
Les classes composants	Fonction
java.lang.Object | +--java.awt.MenuComponent | +--**java.awt.MenuBar**	Création d'une barre des menus dans la fenêtre.
java.lang.Object | +--java.awt.MenuComponent | +--**java.awt.MenuItem**	Création des zones de sous-menus d'un menu principal de la classique barre des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

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

Programme Java

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

Ci-dessous la fenêtre affichée par le programme précédent :



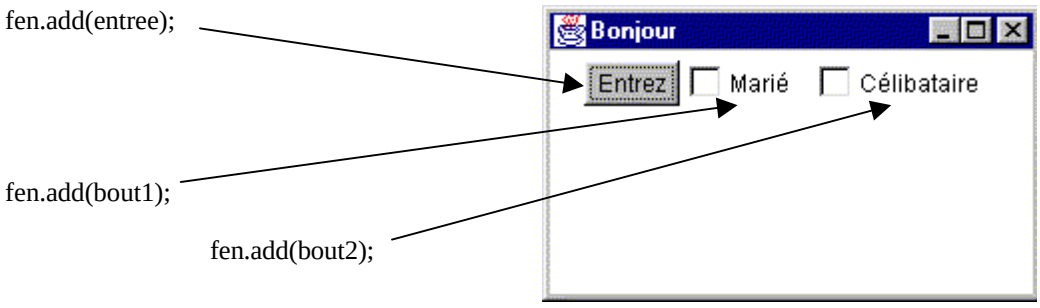
Remarques sur le programme précédent :

1) Les instructions

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

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

2) Les instructions



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

3) L'instruction

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

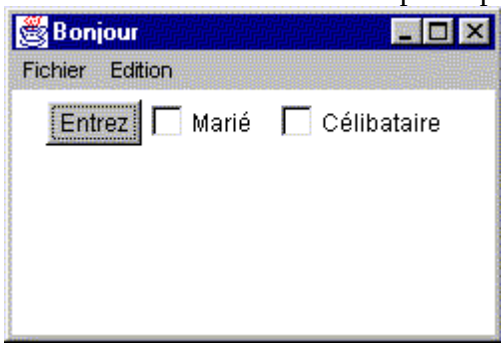
sert à **positionner** les objets visuellement dans la fenêtre les uns à côté des autres, nous en dirons un peu plus sur l'agencement visuel des composants dans une fenêtre.

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

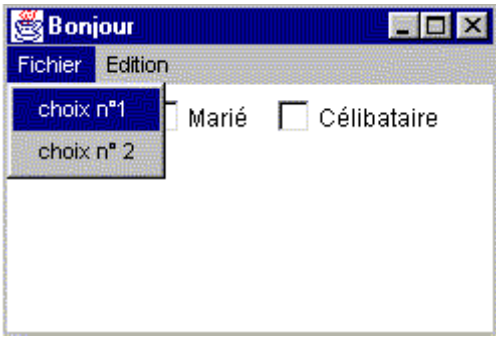
Programme Java

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

Ci-dessous la fenêtre affichée par le programme précédent :

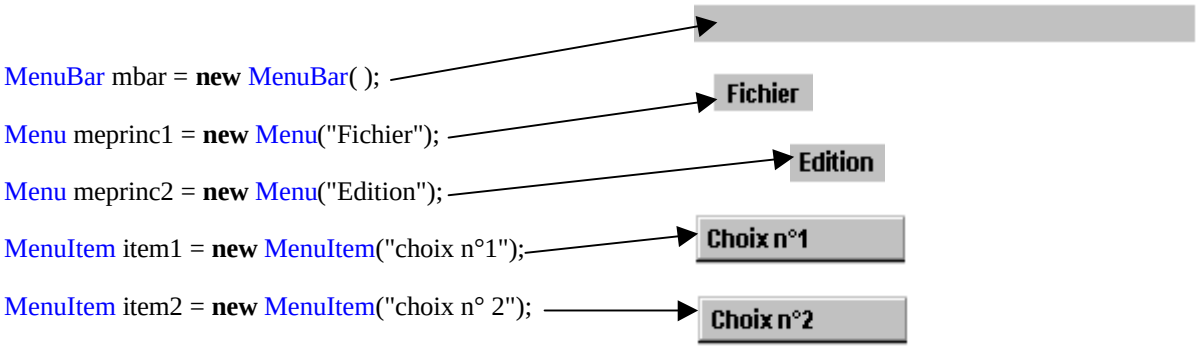


La fenêtre après que l'utilisateur clique sur le menu Fichier



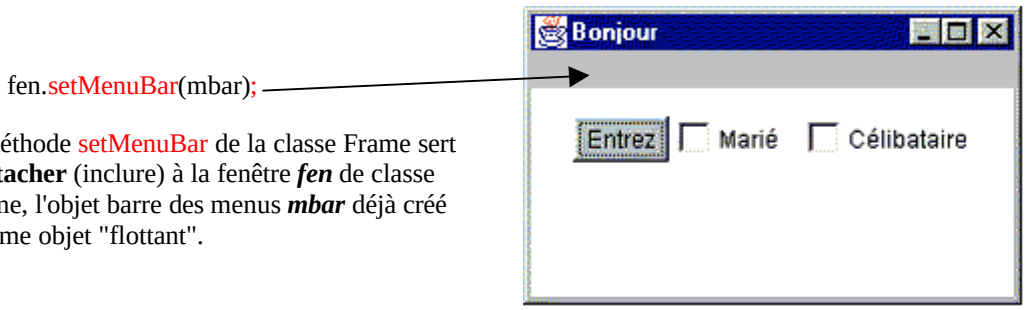
Remarques sur le programme précédent :

1) Les instructions



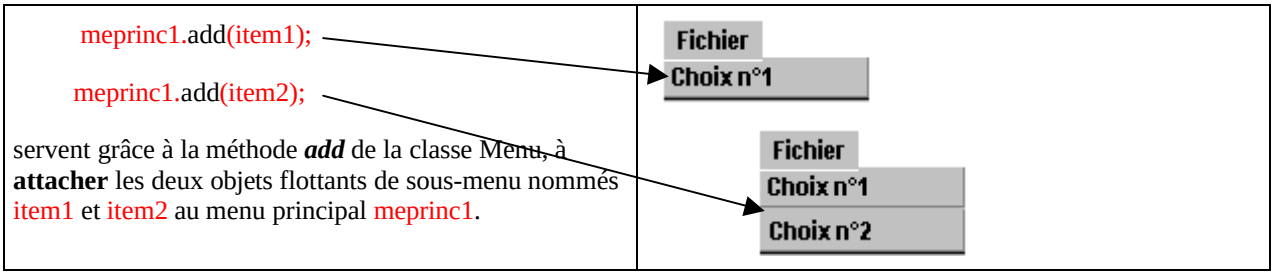
servent à **créer** une barre de menus nommée ***mbar***, deux menus principaux ***meprinc1*** et ***meprinc2***, et enfin deux sous-menus ***item1*** et ***item2*** A cet instant du programme tous ces objets existent mais ne sont pas attachés entre eux, ont peut les considérer comme des objets "flottants"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

2) Dans l'instruction



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

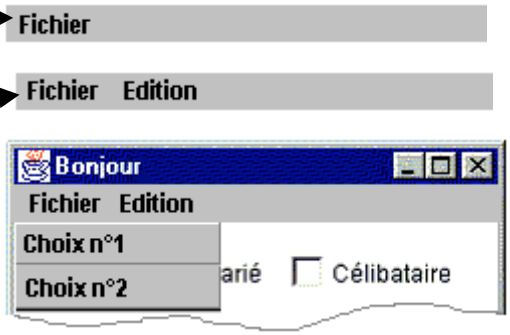
3) Les instructions



4) Les instructions

mbar.add(meprinc1);

mbar.add(meprinc2);



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

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

Composants d'affichage ou de saisie

Composants permettant l'affichage ou la saisie :

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

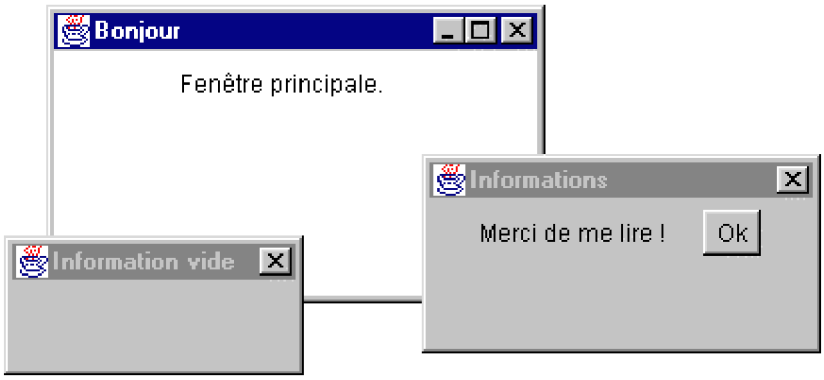
Ces composants s'ajoutent à une fenêtre après leurs créations, afin d'être visible sur l'écran comme les composants de Button, de CheckBox, etc...

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

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

Exemple récapitulatif :

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



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

Première classe :

La classe de dialoque personnalisée

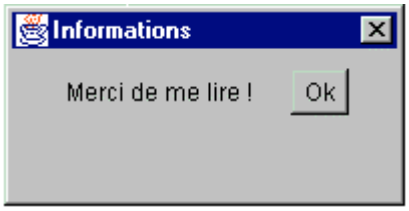
```
import java.awt.*;
class  UnDialog extends Dialog
{
  public UnDialog(Frame mere)
  {
   super(mere,"Informations");
```

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

Explications pas à pas des instructions :

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

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

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

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

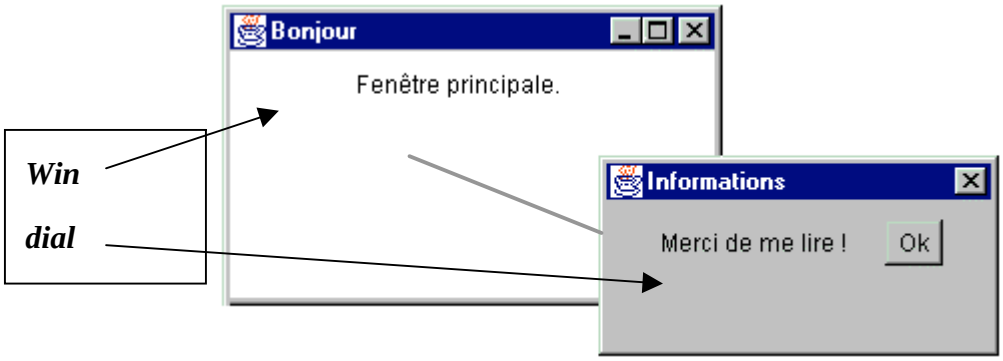
La classe principale contenant main

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

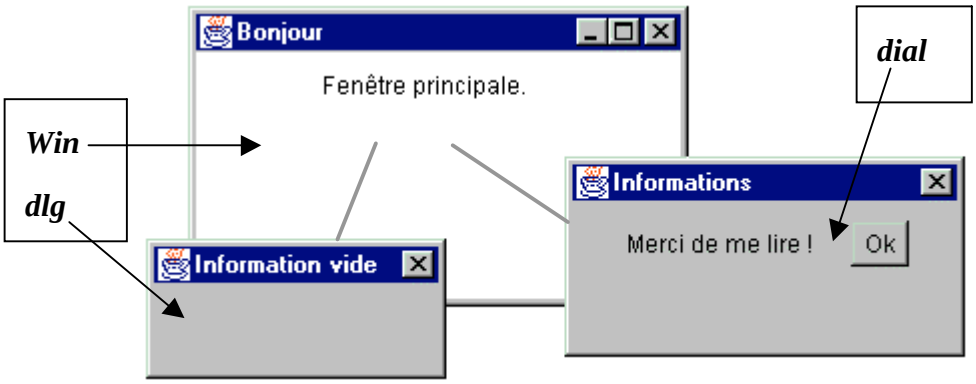
```
}
```

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

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



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



Le programme Java avec les 2 classes

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

```
}
```

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

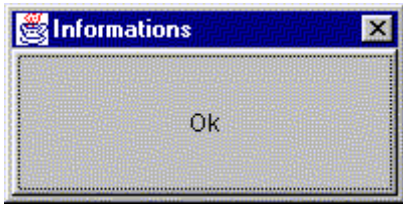
En reprenant la fenêtre de dialogue précédente, observons l'effet visuel produit par la présence ou non d'un **Layout Manager** :

La classe de dialogue sans Layout Manager

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

Voici ce que donne l'exécution de ce programme Java.

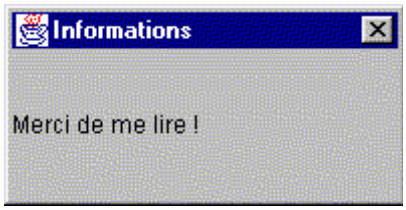
En fait lorsqu'aucun Layout manager n'est spécifié, c'est par défaut la classe du Layout <BorderLayout> qui est utilisée par Java. Cette classe n'affiche qu'un seul élément en une position fixée.



Nous remarquons que le bouton masque l'étiquette en prenant toute la place.

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

voici l'effet visuel obtenu :



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

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

La classe de dialogue avec FlowLayout

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

voici l'effet visuel obtenu :

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

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

on obtient l'affichage inversé :

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

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

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

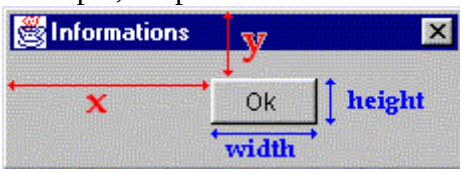
Vous notez qu'il est impossible d'être exhaustif sans devenir assommant, à chacun d'utiliser les Layout en observant leurs effets visuels.

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

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

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

Exemple, les paramètres de setBounds pour un Button :

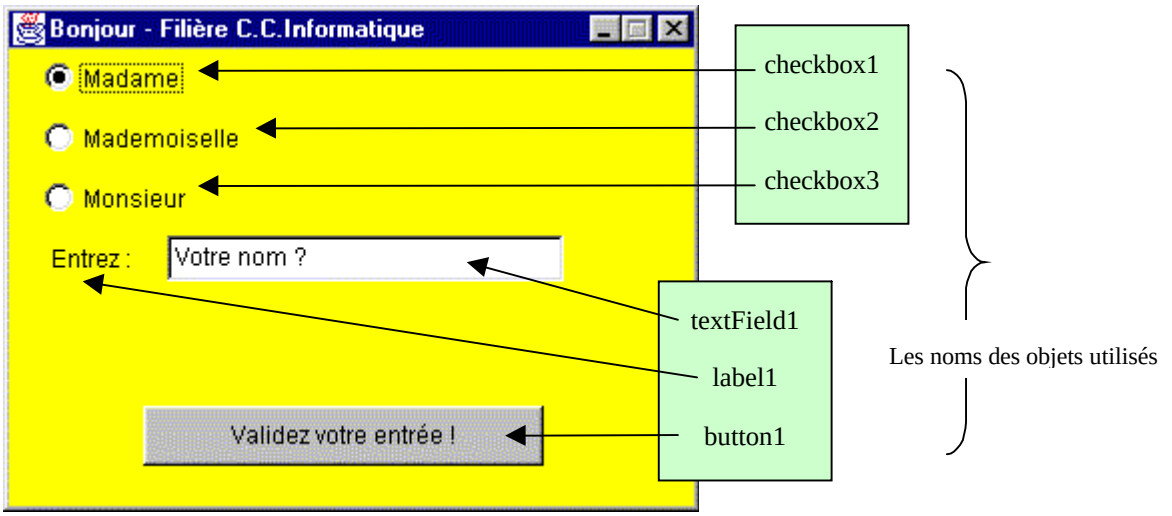


Si nous voulons positionner nous mêmes un composant ***Component*** *comp* dans la fenêtre, nous utiliserons la méthode add indiquant le genre de façon de ranger ce composant (LayoutManager)

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

Une application fenêtrée pas à pas

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



ci-après le code Java du programme :

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

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

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

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

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

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

Les événements

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

Principes de la programmation par événements

La programmation événementielle :

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

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

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

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

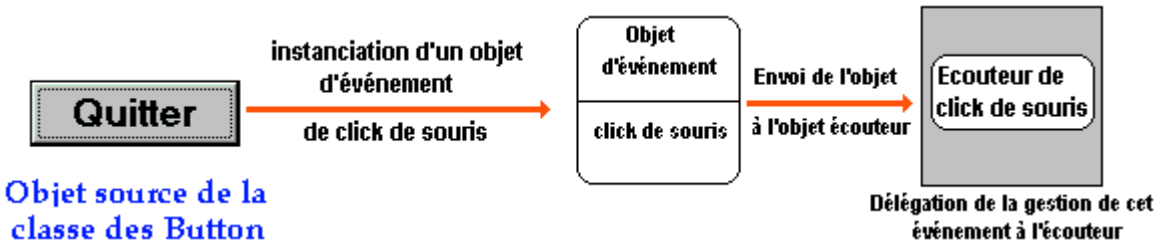
La combinaison de ces deux approches produit un logiciel habillé et adapté au système d'exploitation.

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

Modèle de délégation de l'événement en Java

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

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



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

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

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

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

Intercepter un click de souris sur un bouton

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

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

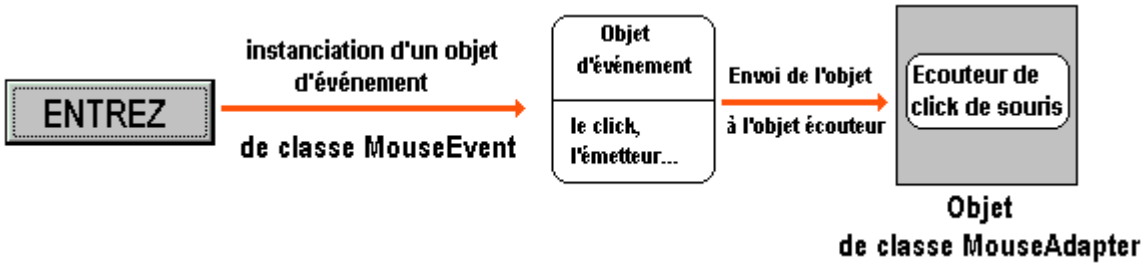
Interception de bas niveau :

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

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

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

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

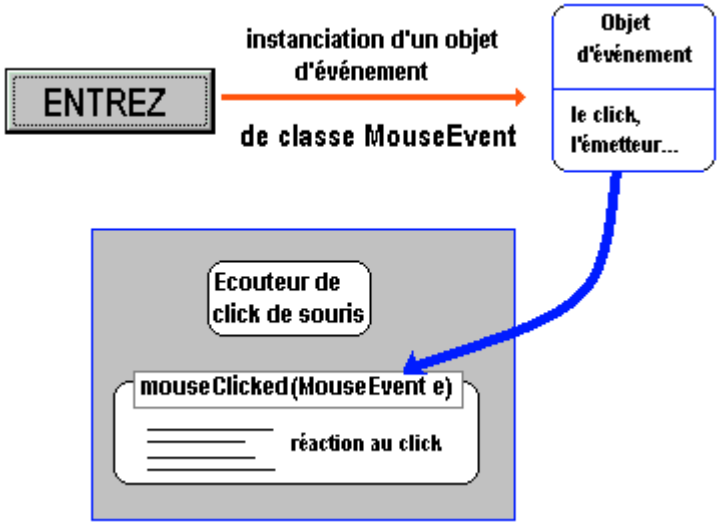


Dans ce cas il suffit de redéfinir la méthode de la classe MouseAdapter qui est chargée d'intercepter et de traiter l'événement qui nous intéresse (cet événement lui est passé en paramètre):

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

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

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



Démarche pratique pour gérer le click du bouton

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

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

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

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

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

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

Interception de haut niveau ou sémantique :

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

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

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

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

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

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

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

Version avec une classe implémentant ActionListener

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

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

Version avec une classe anonyme

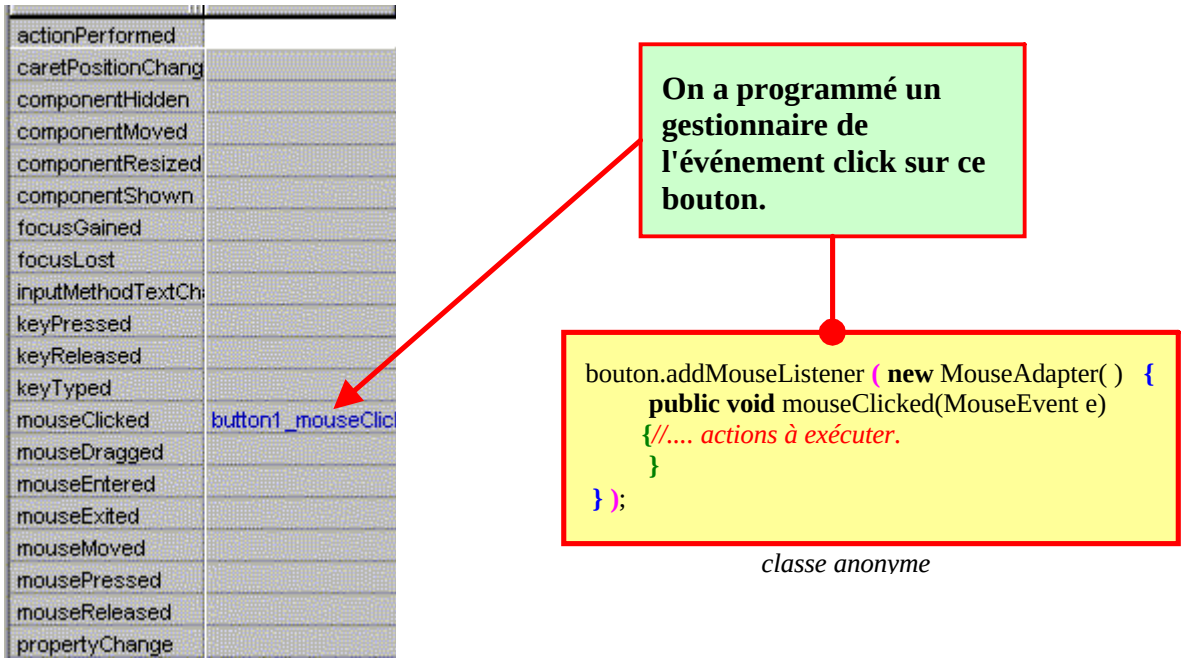
```
  import java.awt.*;
  import java.awt.event.*;

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

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

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

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

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



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

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

Terminer une application par un click de bouton

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

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

Classe dérivée de ActionListener

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

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

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

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

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

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

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

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

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

Méthode main

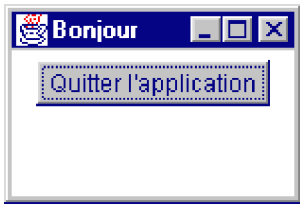
```
public static void main(String [] arg) {
    Frame fen = new Frame ("Bonjour" );
    fen.setBounds(100,100,150,80);
    fen.setLayout(new FlowLayout( ));
    Button quitter = new Button("Quitter l'application");
    quitter.addActionListener(new ListenerQuitter( ));
    fen.add(quitter);
    fen.setVisible(true);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

Le programme Java complet

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

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

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

La fenêtre associée à ce programme :

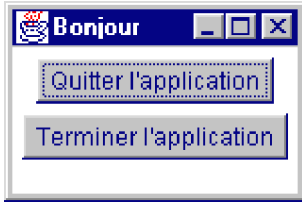


Voici une version de la méthode main du programme précédent dans laquelle nous affichons un deuxième bouton "Terminer l'application" auquel nous avons ajouté le même gestionnaire de fermeture de l'application :

Méthode main

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

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



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

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

La classe anonyme remplaçant tout le code :

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

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

Intérêt d'implémenter une interface XXXListener

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

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

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

Soit au départ l'interface suivante :

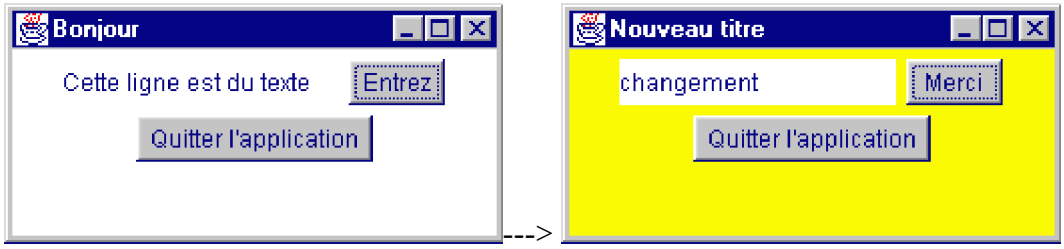
Nous programmons :

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

Le programme Java complet

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

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

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

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



Intérêt d'hériter d'une classe XXXAdapter

Fermer une fenêtre directement sans passer par un bouton

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

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

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

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

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

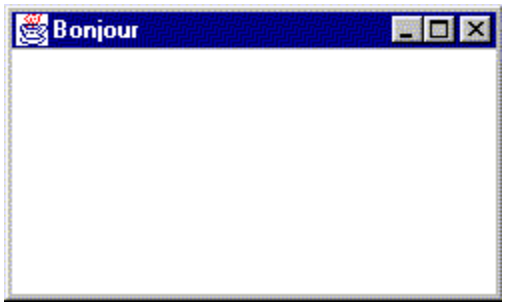
Dans notre cas c'est la méthode **void** windowClosing(WindowEvent e) qui nous intéresse, puisque nous souhaitons terminer l'application à la demande de fermeture de la fenêtre.

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

Version avec une classe anonyme

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

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

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

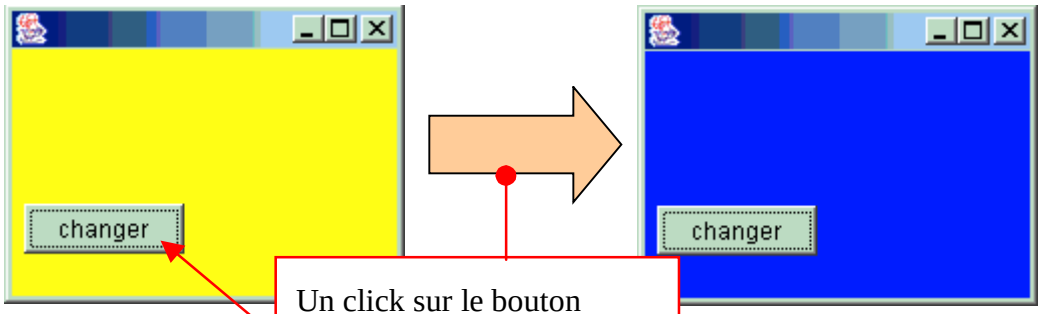


Exercices

Java2 IHM - Awt

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

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



Un click sur le bouton "**changer**", fait changer la couleur du fond de la fiche.

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

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

  class  ecouteur  extends  MouseAdapter    {

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

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

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

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

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

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

class  ecouteur  extends  MouseAdapter   {
 private  Fenetre fenLocal ;

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

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

class  Fenetre  extends  Frame   {

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

public class  ExoAwt   {

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

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

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

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

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

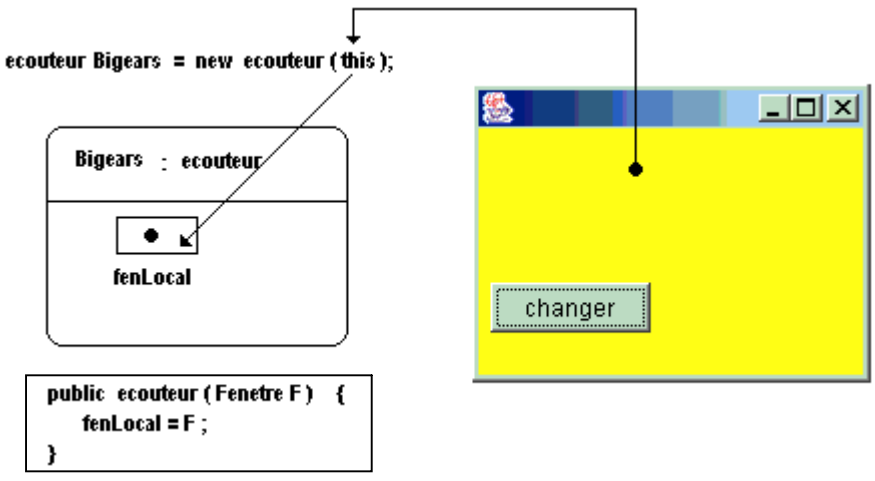


fig - schéma d'accès à la fiche par l'écouteur de classe externe

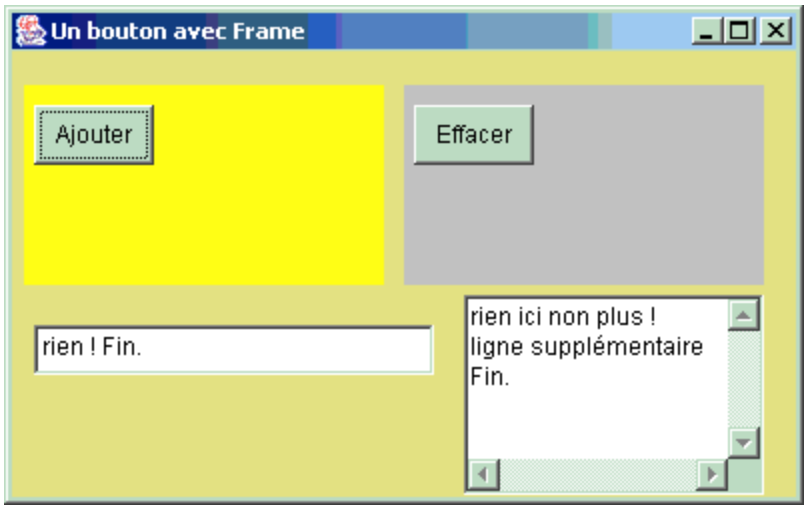
Voici la version la plus courte en code, version conseillée lorsque l'on n'a pas de travail particulier à faire exécuter par un écouteur et que l'on n'a pas besoin d'utiliser la référence de cet écouteur. Cette version utilise la notion de classe anonyme qui est manifestement très adaptée aux écouteurs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

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

class  Fenetre  extends  Frame    {

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

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

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

public class  ExoAwtAnonyme    {

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

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

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

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



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

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

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

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

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

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

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

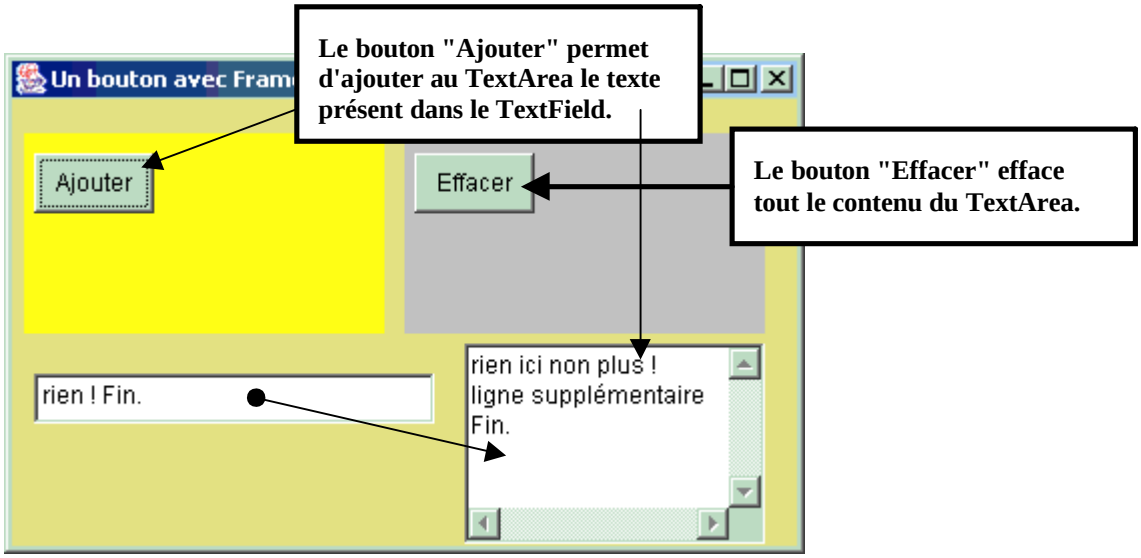
```
this.addWindowListener (   new  WindowAdapter ( )
  {
       public void  windowClosing ( WindowEvent e )  {
         System.exit ( 0 );
       }
   }
 );
```

L'écouteur gère le windowClosing**.**

Evénéments-2 :

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

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



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

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

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

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

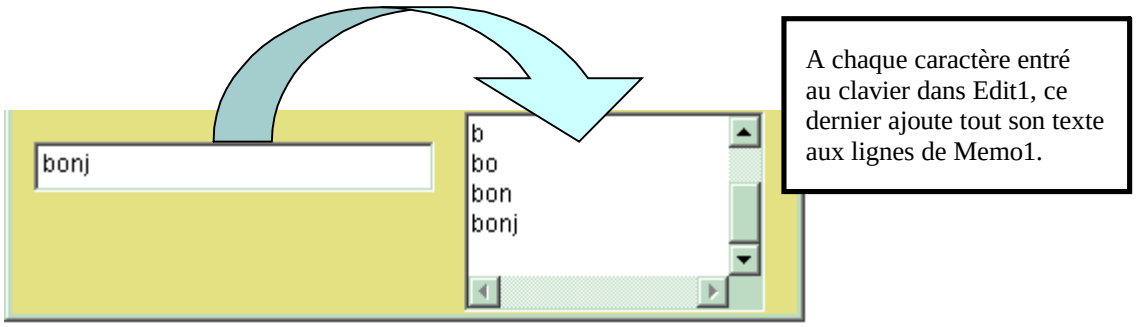
Classe anonyme d'écouteur dérivant des MouseAdapter**.**

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

L'écouteur gère le mouseClicked

Evénement-3 :

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



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

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

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

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

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

L'écouteur gère le textValueChanged

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

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

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

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

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

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

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

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

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

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

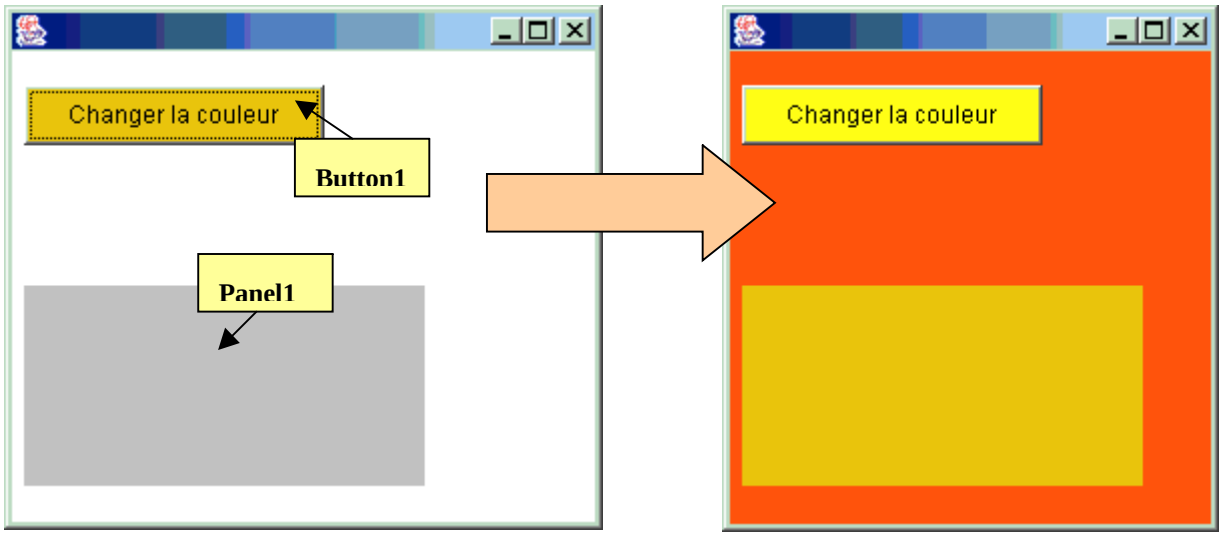
IHM - Awt : Variations de souris sur une fenêtre et écouteur centralisé

Deux versions d'une même IHM

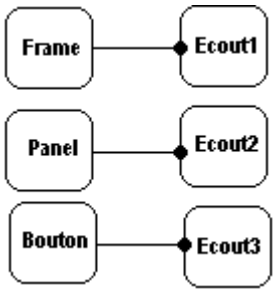
Soit l'IHM suivante composée d'une fiche de classe **Frame**, d'un bouton Button1 de classe Bouton dérivée de la classe Button , et d'un panneau Panel1 de classe **Panel**.

L'IHM réagit uniquement au click de souris :

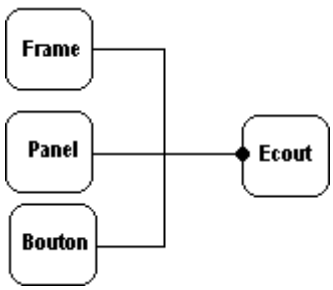
- Le Button1 de classe Bouton réagit au simple click et il fait alternativement changer de couleur le fond de la fiche sur laquelle il est déposé.
- Le Panel1 de classe Panel réagit au simple click et au double click, chaque réaction click ou double-click fait changer sa couleur de fond.
- La fiche de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



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



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



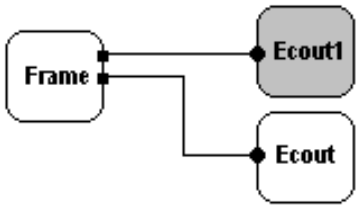
Nous pouvons aussi instancier un écouteur général (**centralisé**) qui écoutera les 3 objets.

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

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

class  AppliUneFrameClick2
{

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

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

Si l'émetteur du MouseEvent est du type Fenetre.

Si l'émetteur du MouseEvent est du type Panneau.

Si l'émetteur du MouseEvent est du type Bouton.



La fenêtre possède deux écouteurs :

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

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

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

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

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

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

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

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

La fiche héritant de Frame avec ses composants déposés.

L'écouteur centralisé est recensé auprès des 3 objets.

La fiche this recense son écouteur pour WindowClosing

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

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

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

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

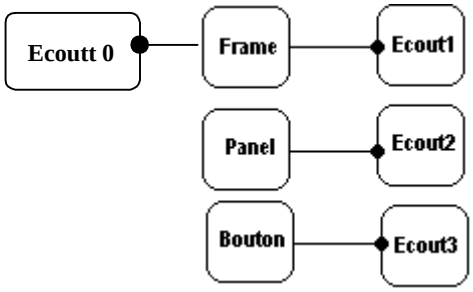
Accès à la fiche comme parent

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

Accès à la fiche par une référence

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

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

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




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

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

class  AppliUneFrameClick3
{

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


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

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

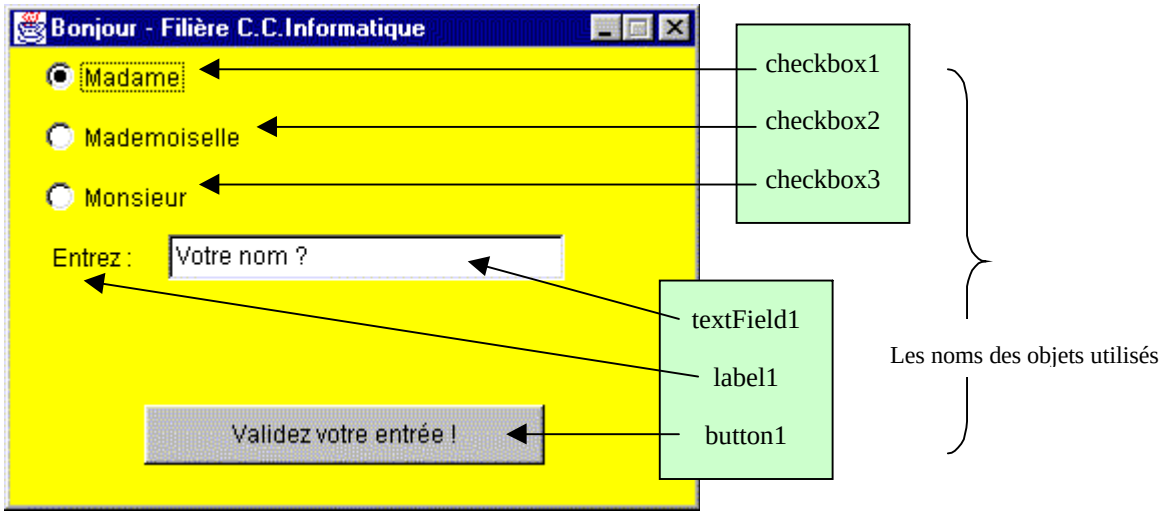
Classes anonymes héritant de MouseAdapter

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

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

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

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

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

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

Identique au code de la version précédente

IHM - Awt : Saisie de renseignements interactive

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



Description événementielle de l'IHM :

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

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

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

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

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

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

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

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

Le texte du textField1 a changé

Click dans l'un des checkBox

Click sur le button1 "valider…"

IHM - Awt : Fermer une Frame directement par processWindowEvent

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

ProcessWindowEvent :

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

protected void processWindowEvent(**WindowEvent** e)

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

public abstract class **AWTEvent**

static long ACTION_EVENT_MASK
static long ADJUSTMENT_EVENT_MASK
….

static long WINDOW_EVENT_MASK
static long WINDOW_FOCUS_EVENT_MASK
static long WINDOW_STATE_EVENT_MASK

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

Class WindowEvent

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

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

public int getID ( )

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

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

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

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

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

```
class Fenetre extends Frame {

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

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

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

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

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

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

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

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

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

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

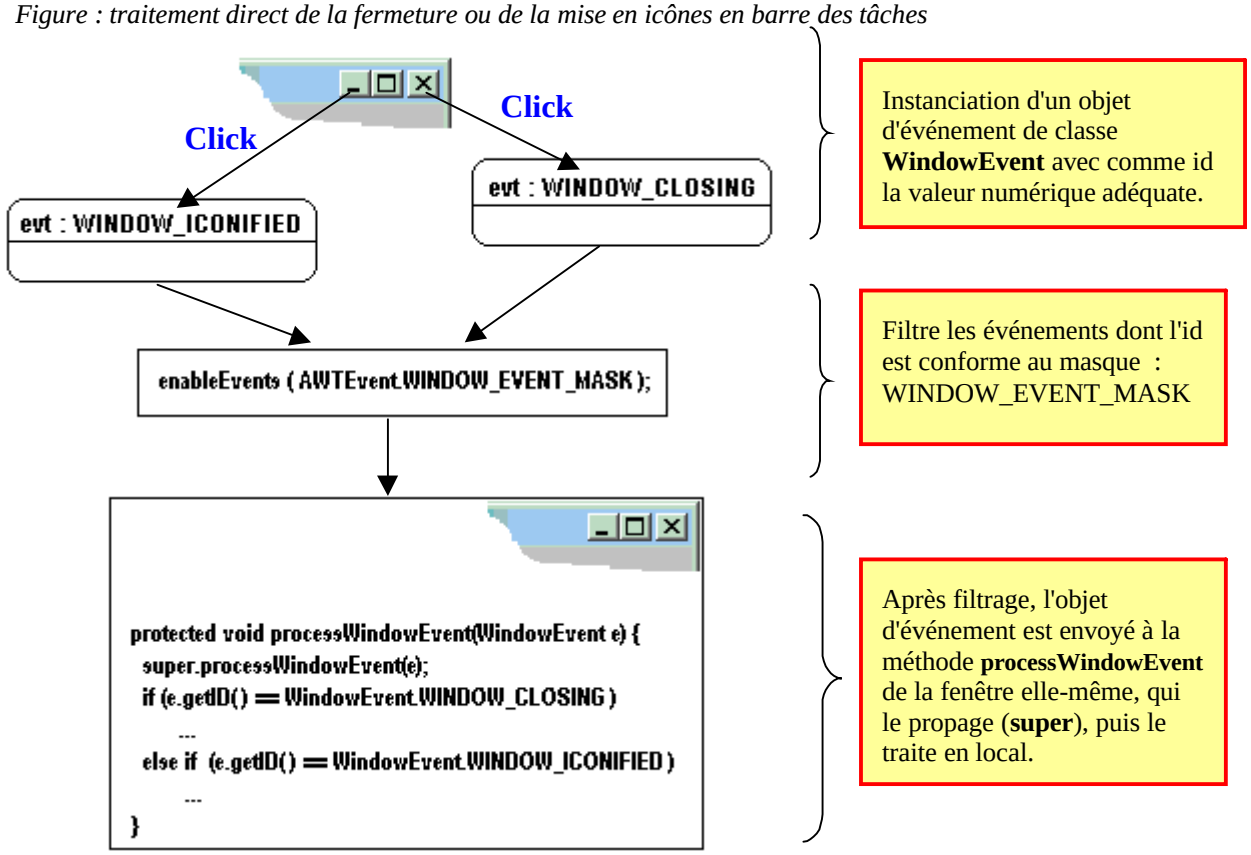
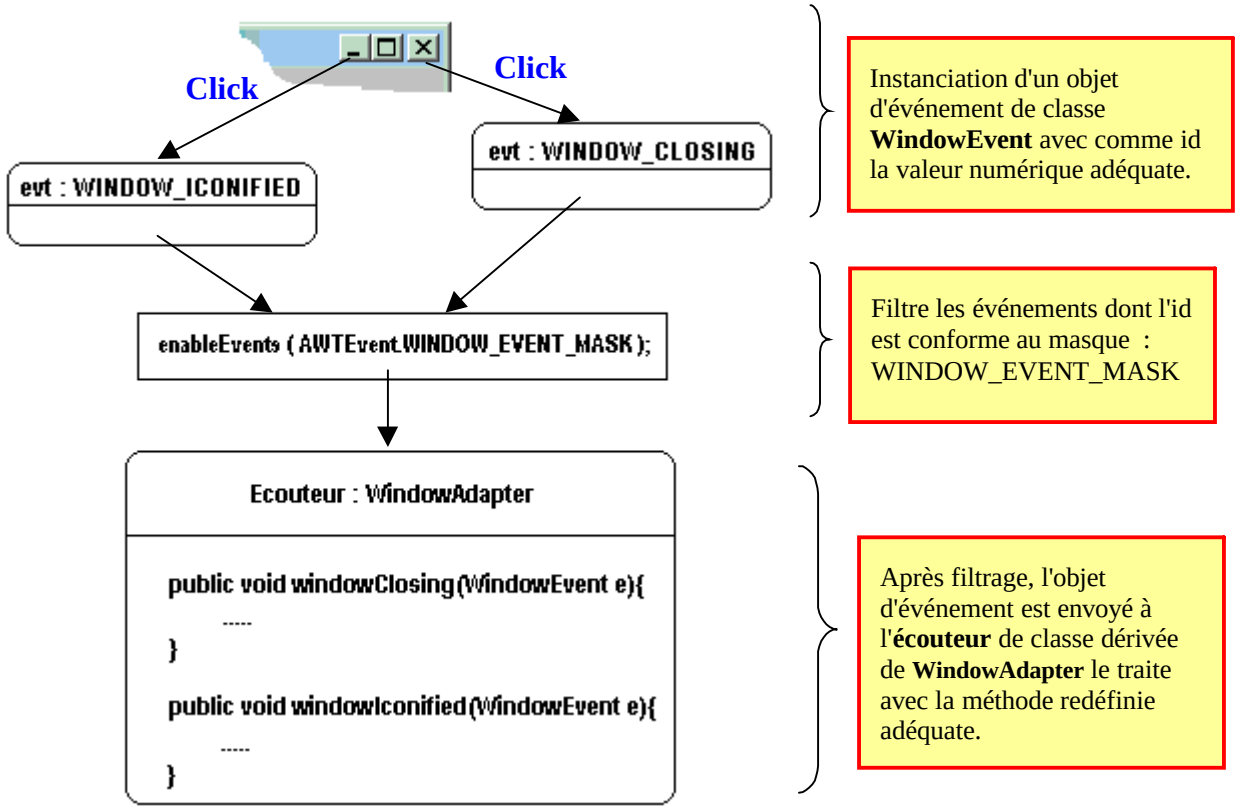


Figure : traitement par écouteur de la fermeture ou de la mise en icônes en barre des tâches

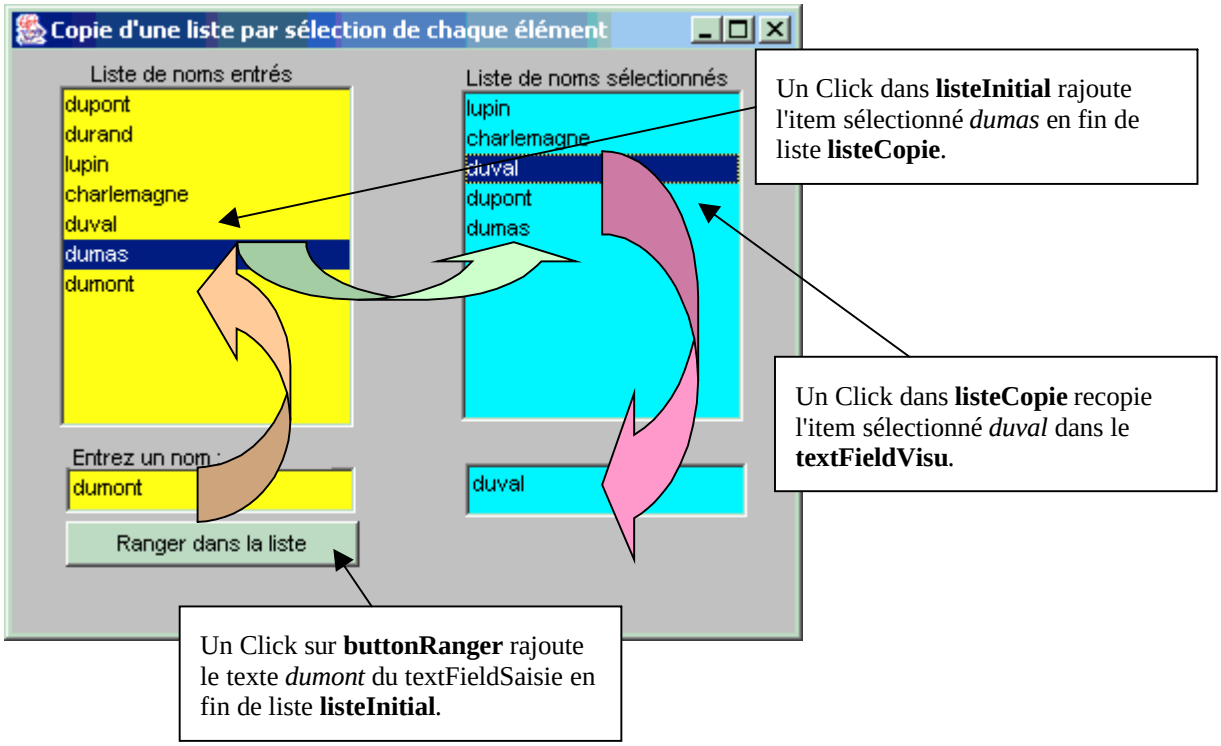


IHM - Awt : Utiliser une java.awt.List

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

L'IHM réagit uniquement au click de souris :

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



Code Java de la classe Fenetre

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

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

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

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

Classe anonyme

Classe anonyme

Classe anonyme

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

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

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

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

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

Fermeture de la fenetre sur l'événement : WINDOW_CLOSING

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

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

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

IHM - avec Swing

Java2

Composants lourds, composants légers

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

Les composants lourds

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

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

Tous les composants du package AWT sont des composants lourds.

Les composants légers

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

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

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

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

Swing contient un minimum de composants lourds

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

Les fenêtres Swing sont des composants lourds

Les fenêtres en Java Swing :

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

Hiérarchie de classe de ces composants de fenêtres :

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

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

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

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

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

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

Les autres composants Swing sont légers

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

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

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

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

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

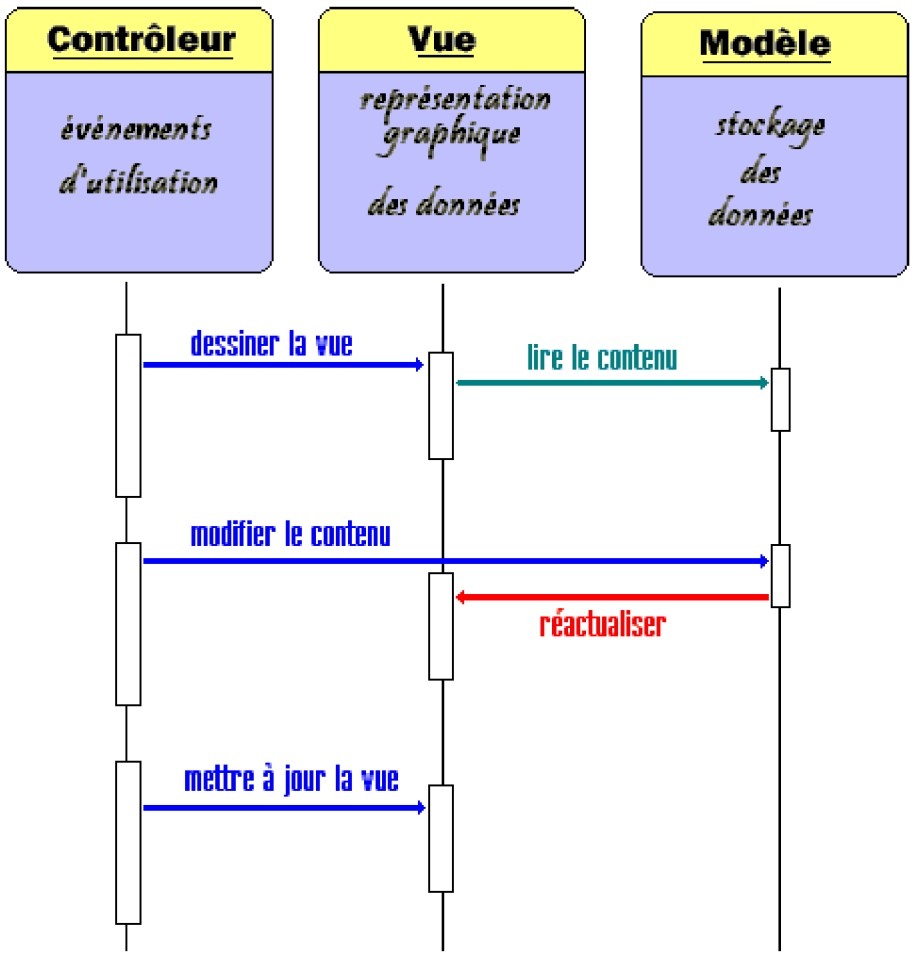
L'architecture Modèle-Vue-Contrôleur en général (MVC)

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

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

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

Diagramme de séquence UML des interactions MVC



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

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

Le pluggable look and feel

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

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

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

Trois exemples de look and feel

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

Lignes de code pour passer en IHM motif :

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

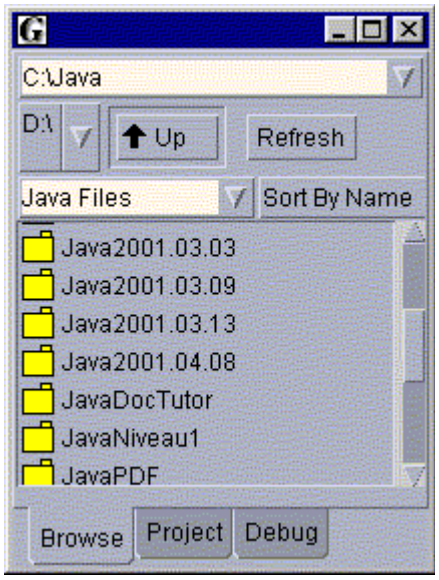
Lignes de code pour passer en IHM métal :

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

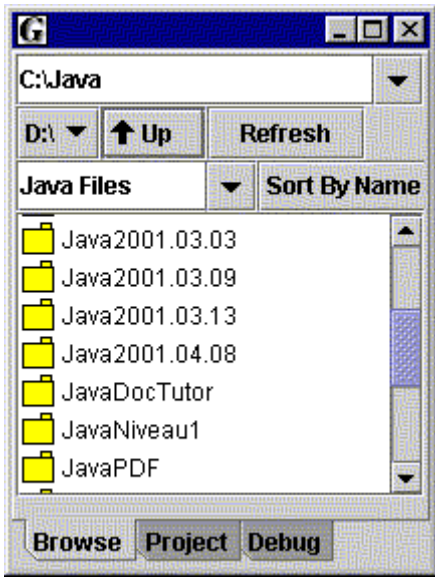
Lignes de code pour passer en IHM Windows :

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

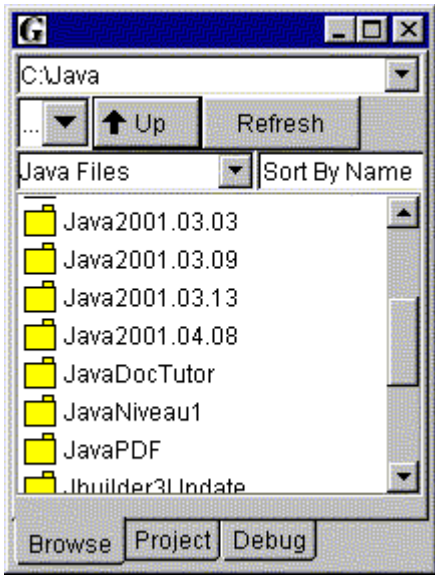
Aspect motif de l'IHM



Aspect métal de l'IHM



Aspect Windows de l'IHM :



Les swing reposent sur MVC

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

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

Exemple de quelques interfaces de modèles rencontrées dans la bibliothèque Swing

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

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

Le composant javax.swing.Jlist

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

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

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

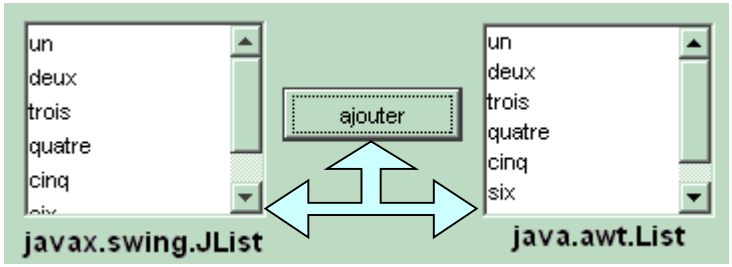
Pour mettre en oeuvre les modèles et les fournir aux composants on utilise la méthode

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

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

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

Comparaison awt.List et swing.JList

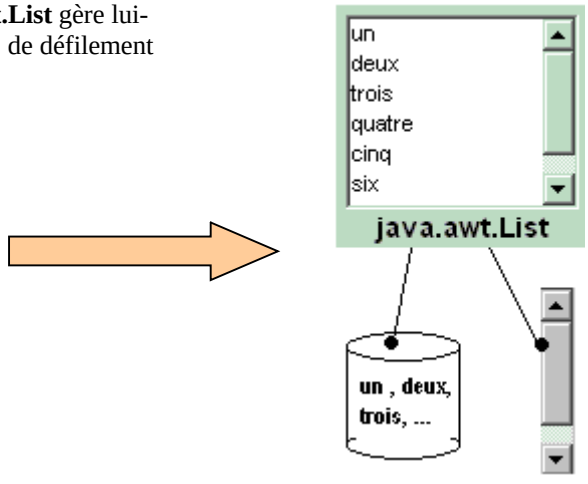


Une IHM dans laquelle, le bouton Ajouter, insère 7 chaînes dans chacun des deux composants.

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

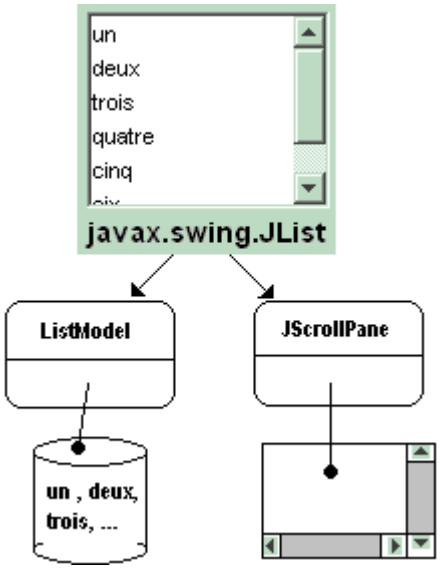
Comme en Delphi le composant **java.awt.List** gère lui-même le stockage des données, et la barre de défilement verticale.

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



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

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



Le composant javax.swing. JTextPane

Soit le code suivant :

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

Caractérisation du style n°1 du document

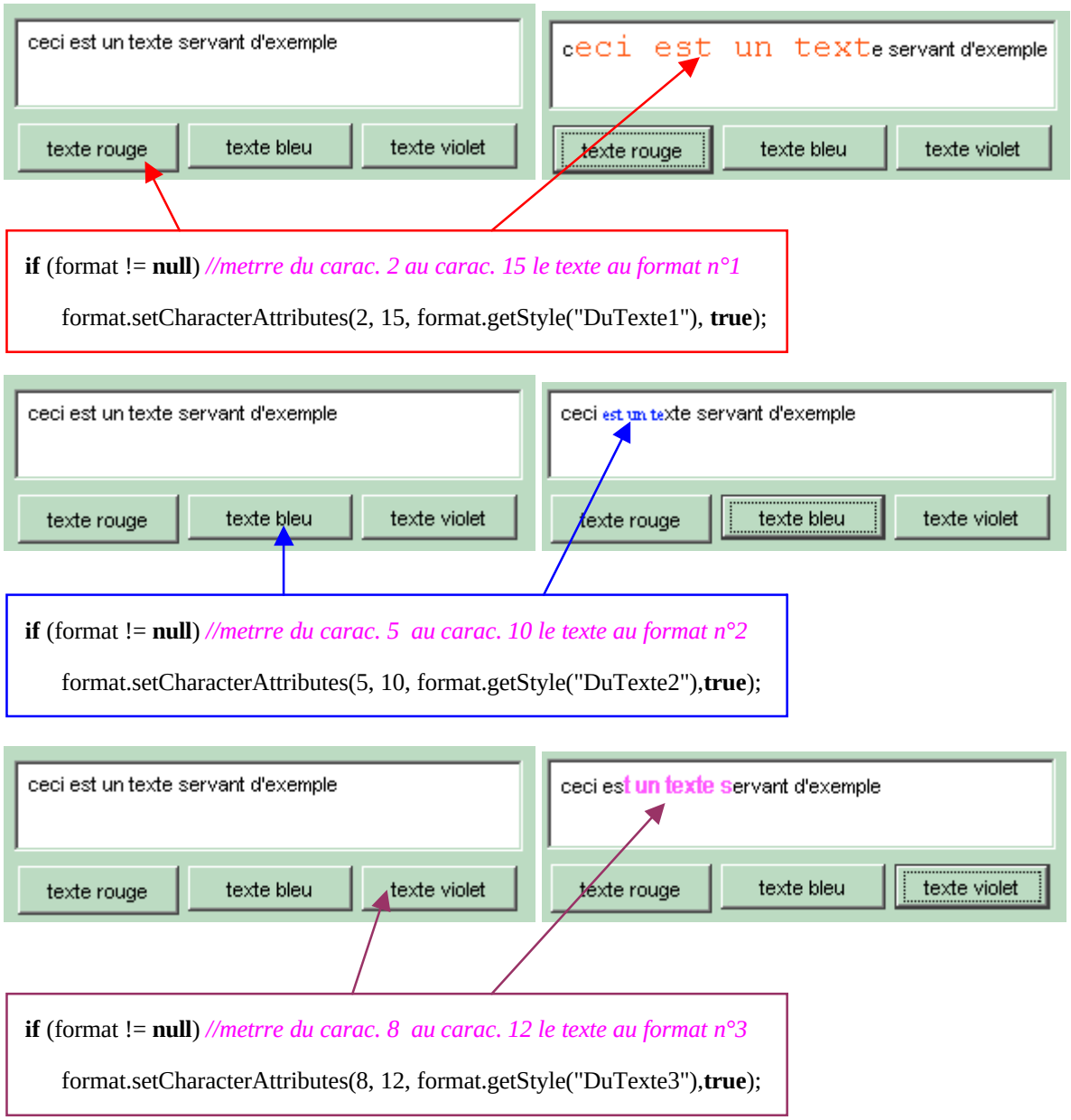
Caractérisation du style n°2 du document

Caractérisation du style n°3 du document

Le document gère les styles du JtextPane.

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

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

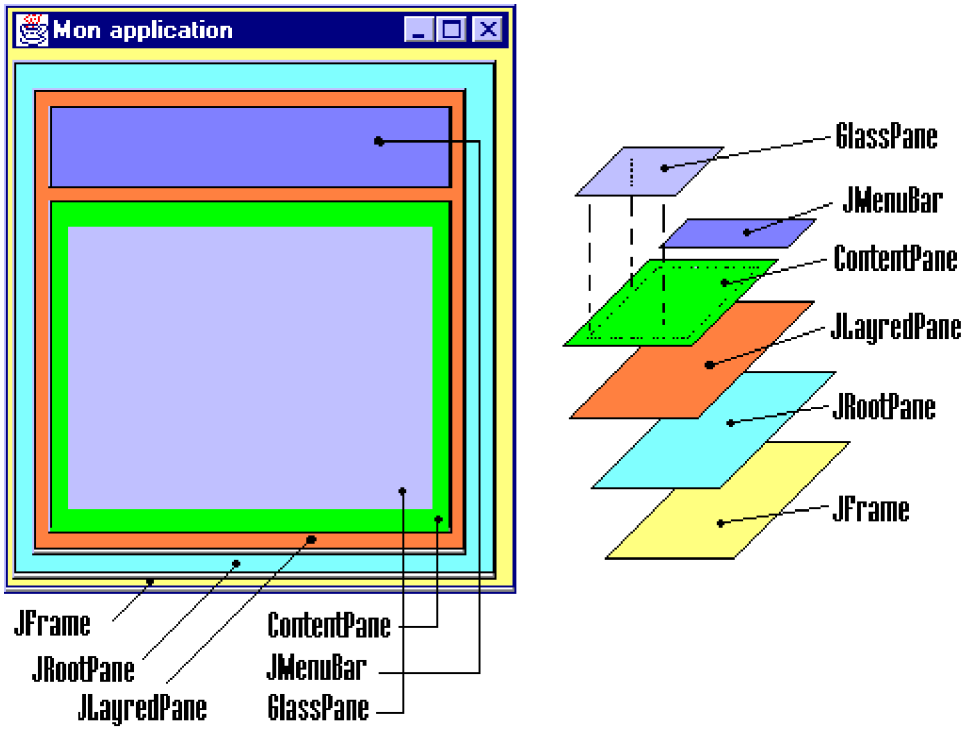


Chaque bouton lance l'application d'un des trois styles d'attributs à une partie du texte selon le modèle de code suivant :

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

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

Système de conteneur pour afficher dans une Fenêtre ou une Applet



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

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

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

Exemple d'ajout d'un bouton à une fenêtre

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

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

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

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

Conseils au débutant

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

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

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

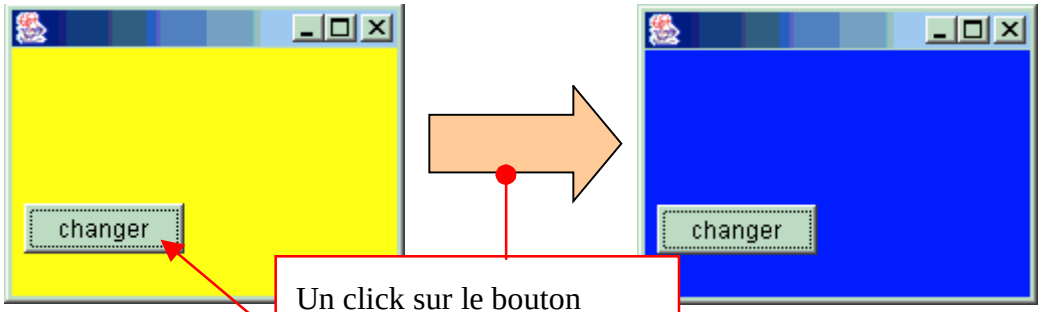
Dans les pages qui suivent nous reprenons les exercices écrits avec les Awt en utilisant leur correspondant Swing. Comme nous avons déjà expliqué les sources nous ne donnerons des indications que lorsque l'utilisation du composant Swing induit des lignes de code différentes.

Exercices IHM - JFrame de Swing

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

L'IHM réagit uniquement au click de souris :

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




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

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

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

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

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

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

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

public class ExoSwing   {

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

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

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

public void setDefaultCloseOperation(**int** operation)

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

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

WindowConstants.DO_NOTHING_ON_CLOSE
WindowConstants.HIDE_ON_CLOSE
WindowConstants.DISPOSE_ON_CLOSE

JFrame.EXIT_ON_CLOSE

Dans la classe : **WindowConstants**

Dans la classe : **JFrame**

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

setDefaultCloseOperation(**JFrame**.EXIT_ON_CLOSE);

Reprise du code de Fenetre avec cette modification spécifique aux JFrame

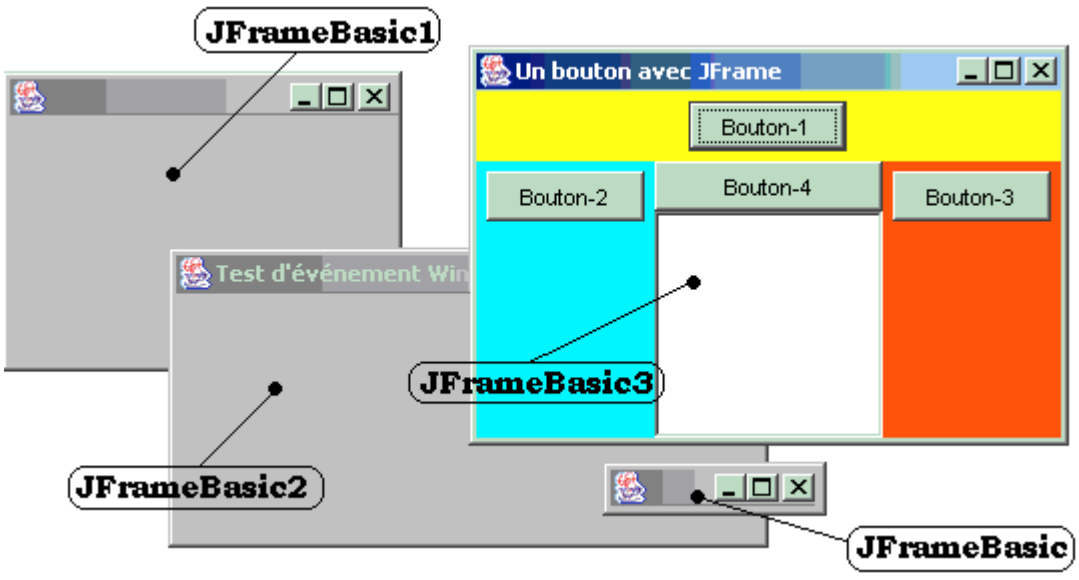
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

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

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

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

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

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

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

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



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

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

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


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

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

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

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

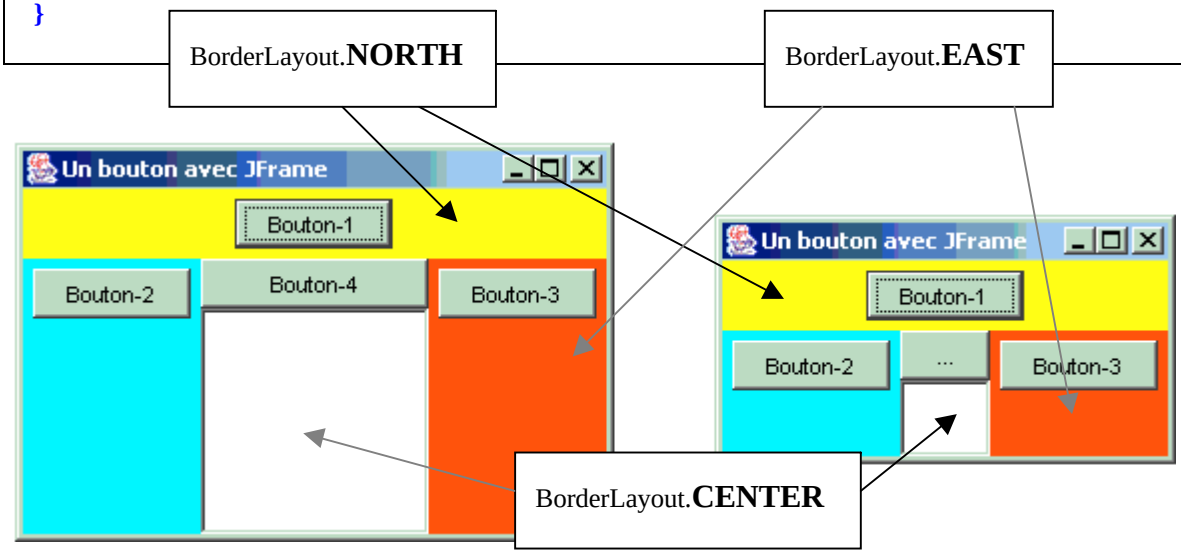
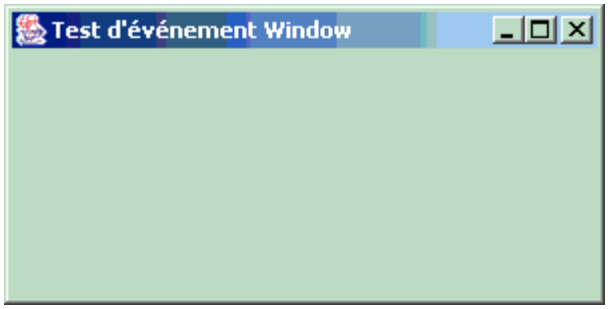


fig-repositionnement automatique des quatre Jpanel grâce au BorderLayout

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

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

En cliquant sur JFrameBasic2 elle passe au premier plan



voici l'affichages console obtenu :

> *JFrameBasic2 / WINDOW_ACTIVATED = 205*

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

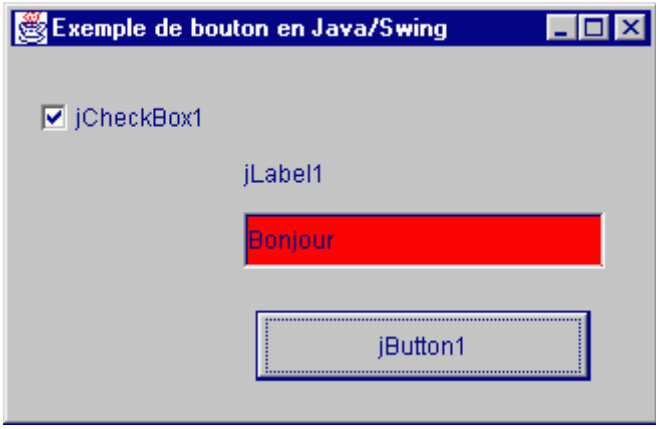
JFrameBasic2 / WINDOW_CLOSING = 201
JFrameBasic2 / WINDOW_DEACTIVATED = 206
JFrameBasic2 / WINDOW_CLOSED = 202 => JFrameBasic2 a été détruite !

Exemple - JButton de Swing

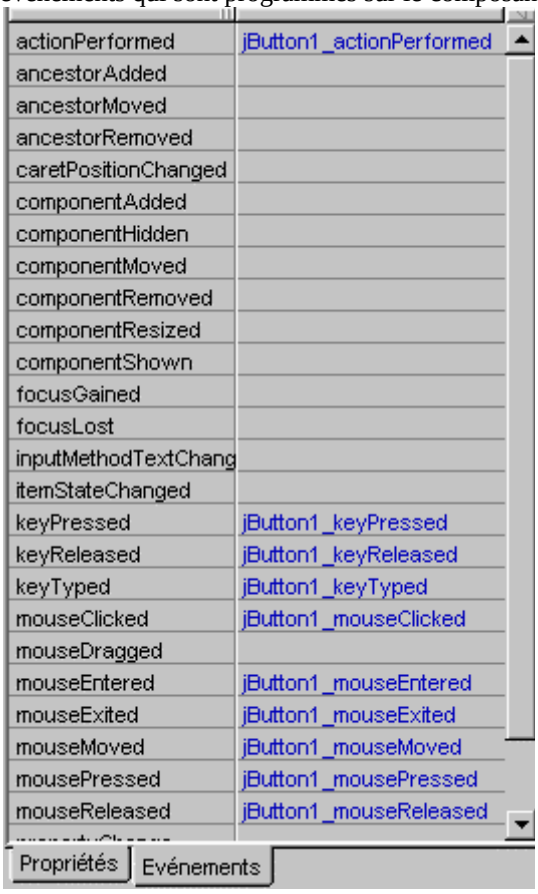
Code java généré par JBuilder

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

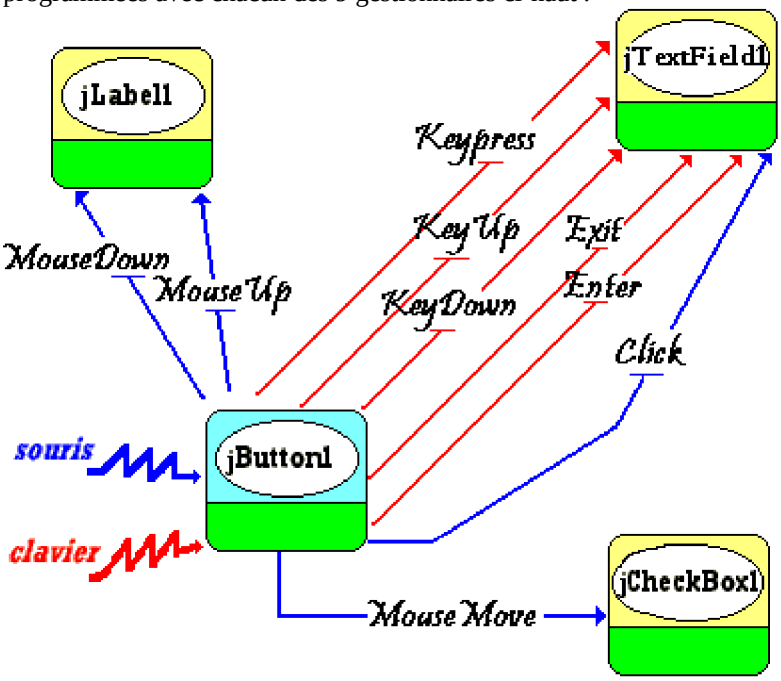
La fenêtre comporte un bouton (JButton jButton1), une étiquette (**JLabel** jLabel1), une case à cocher (**JCheckBox** jCheckBox1) et un éditeur de texte mono-ligne (**JTextField** jTextField1) :



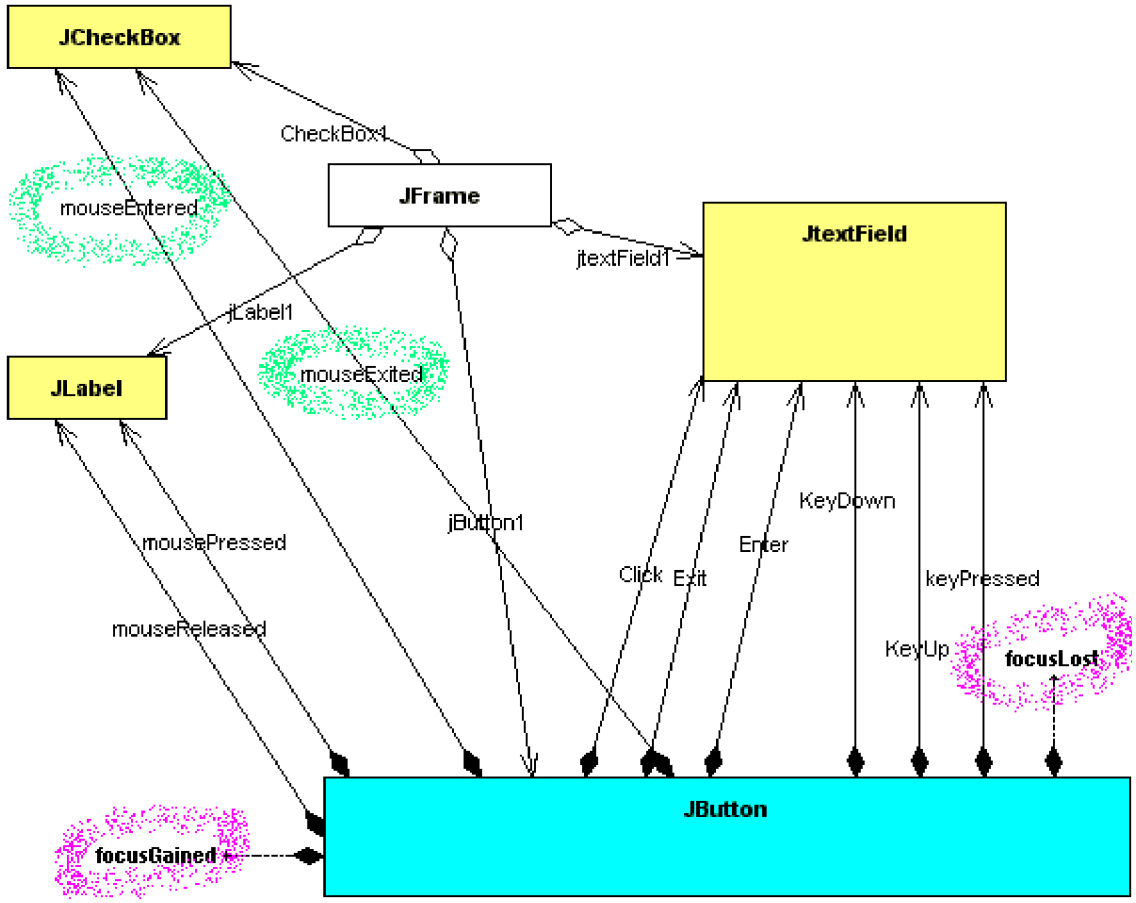
Voici les 10 gestionnaires d'événements qui sont programmés sur le composant jButton de classe **JButton**:



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

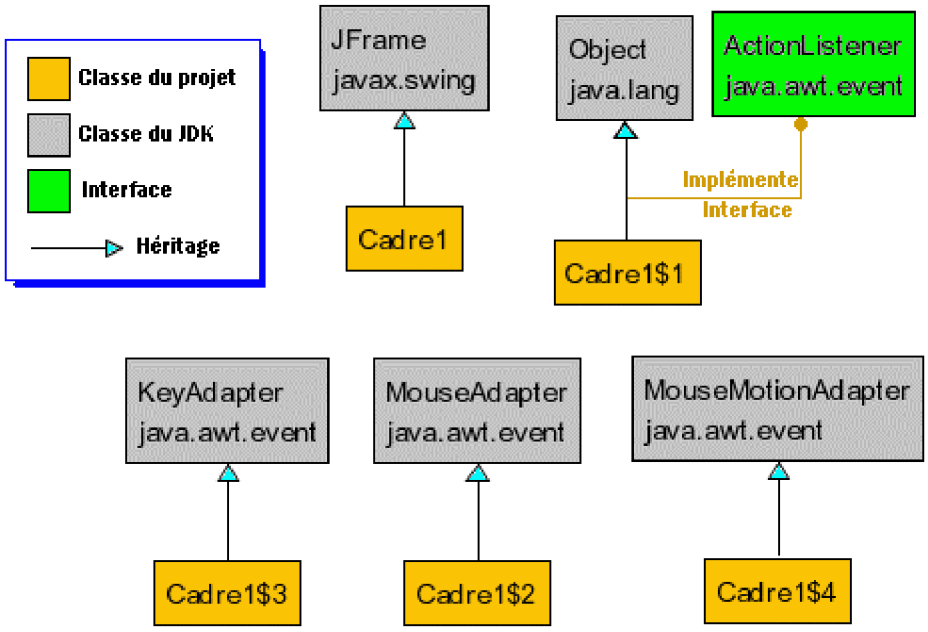


Les actions exit et enter sont représentées en Java par les événements focuGained et focusLost pour le clavier et par les événements mouseEntered et mouseExited pour la souris,. Il a été choisi de programmer les deux événements de souris dans le code ci-dessous.



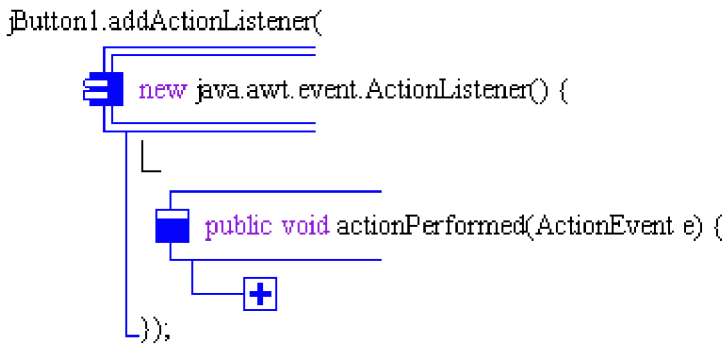
En Java

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

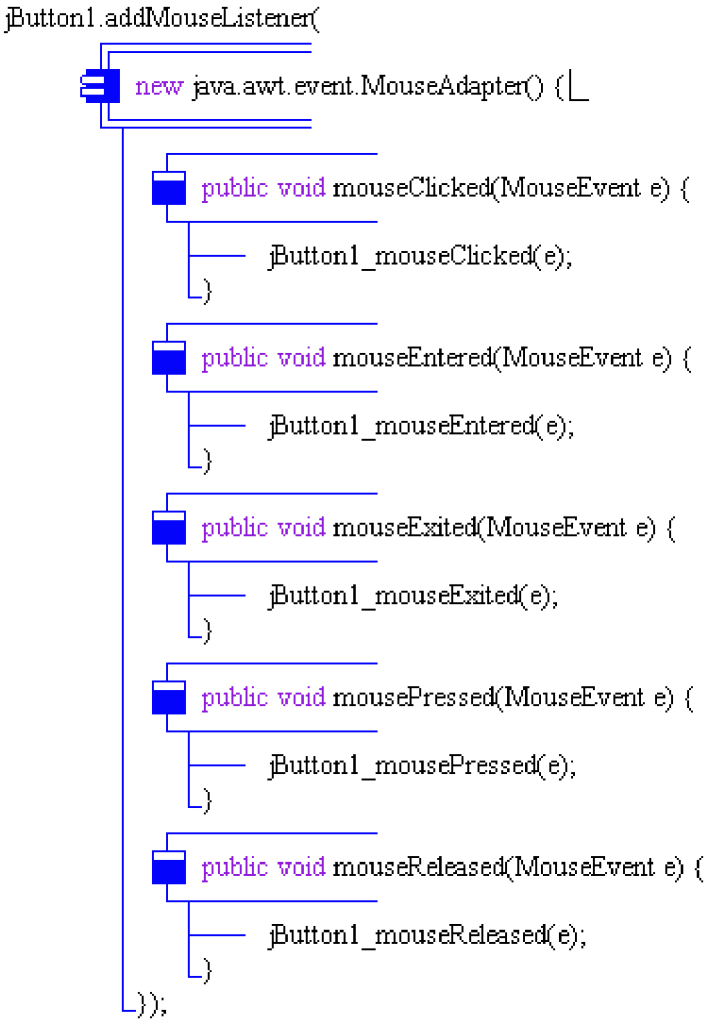


Rappelons que les classes Cadre1$1, Cadre1$2, ... sont la notation des classes anonymes créées lors de la déclaration de l'écouteur correspondant, Java 2 crée donc dynamiquement un objet écouteur interne (dont la référence n'est pas disponible). Ci-dessous les diagrammes jGrasp des quatre classes anonymes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

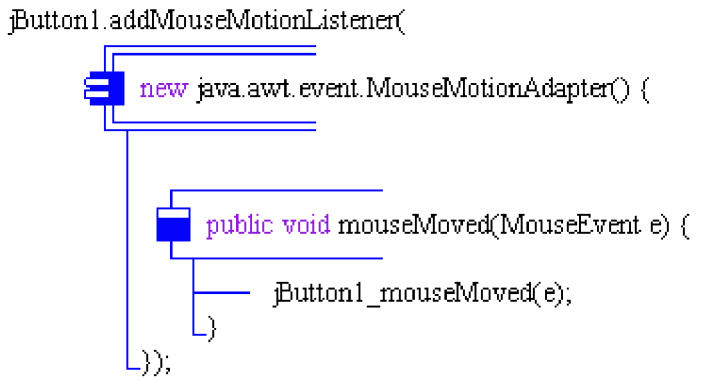
Cadre1$1:



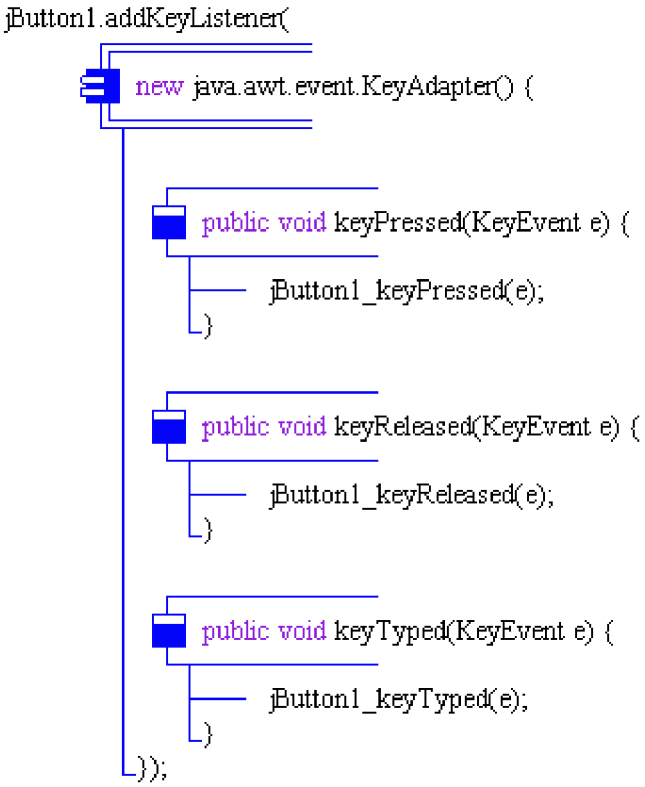
Cadre1$2:



Cadre1$4:



Cadre1$3:



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

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

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

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

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

      public void actionPerformed(ActionEvent e) {
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Attention sur un click de souris l'événement :

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

ou bien **désactivé** :

Attention sur un click de souris l'événement :

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

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

Exemple - JcheckBox , JRadioButton

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

cas d'un seul composant conteneur

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

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

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

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

Cas de plus d'un composant conteneur

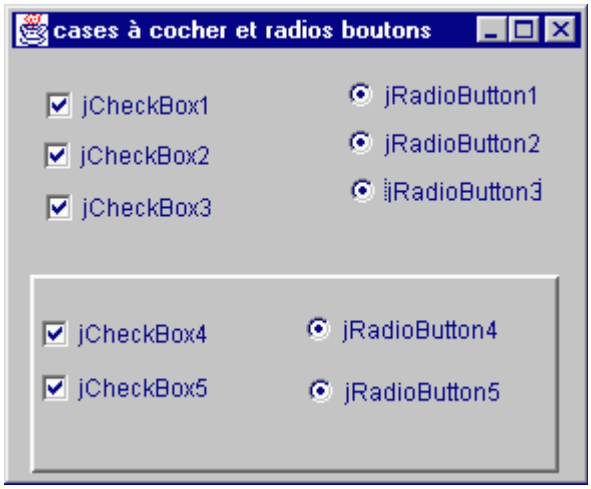
La fenêtre d'exemple comporte :

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

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

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

Voici le résultat obtenu :



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

Regroupement de boutons s'excluant mutuellement

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

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

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

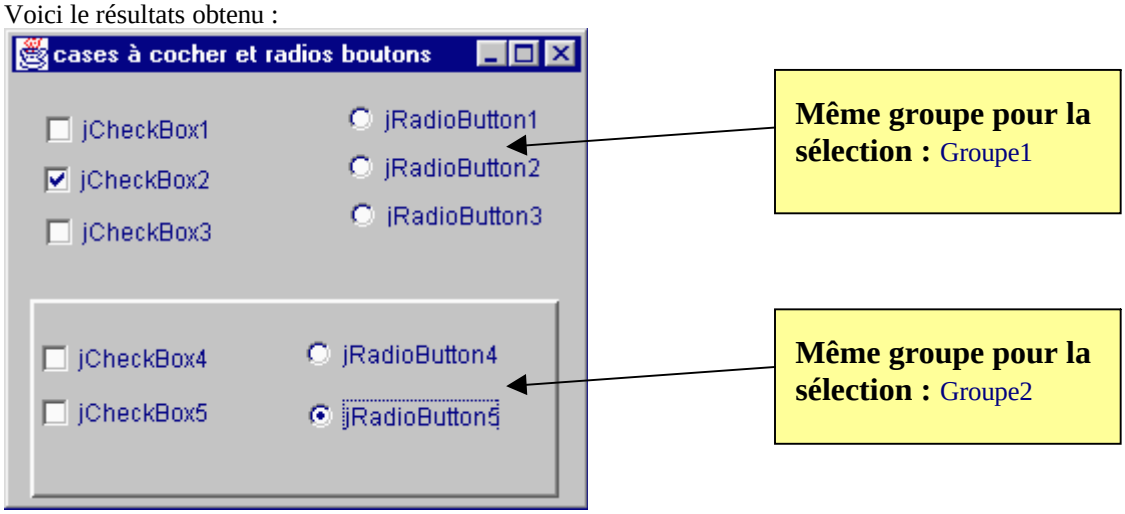
Nous rattachons au Groupe1 tous les composants déposés sur le **JPanel** (jCheckBox4, jCheckBox5, jRadioButton4 et jRadioButton5 font alors partie du même groupe d'exclusion mutuelle) :

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

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

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

Voici le résultats obtenu :



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

Amélioration interactive du JCheckBox sur le Checkbox

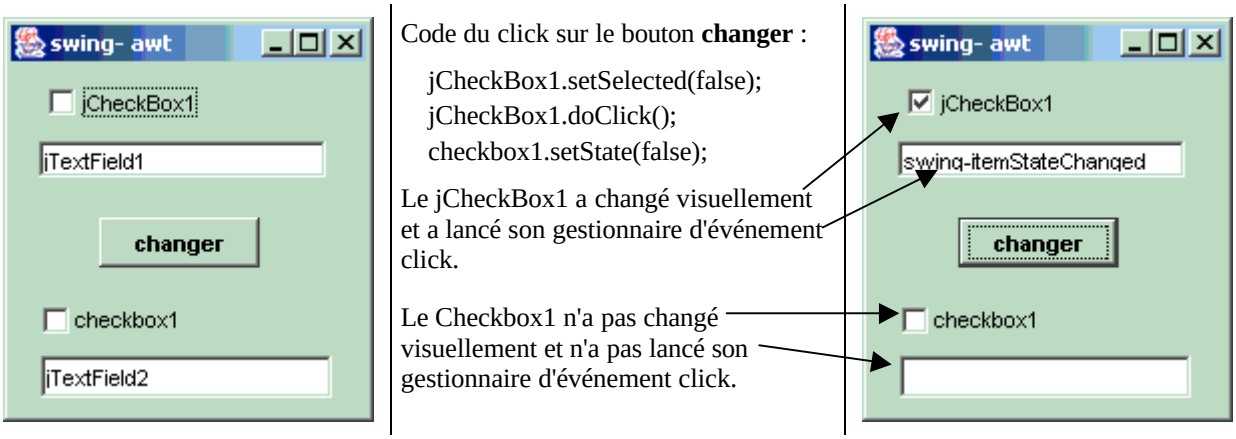
Il est possible dans le code, grâce à une méthode de modifier la valeur propriété cochée ou non cochée :

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

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

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

public void doClick()



Exemple - JComboBox

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

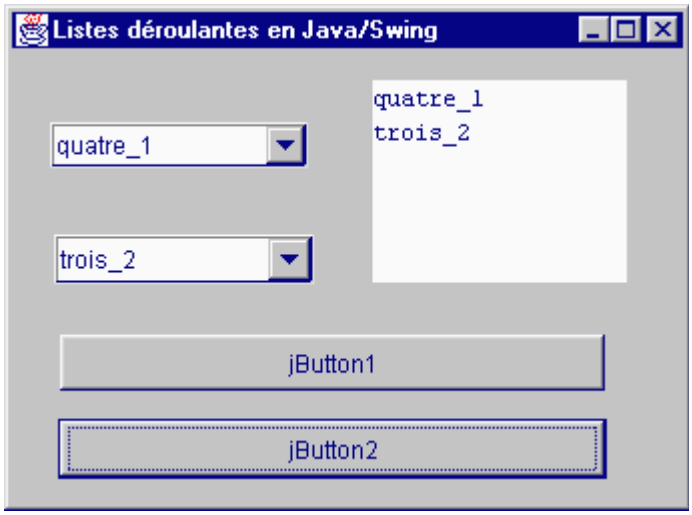
Dans cet exemple, nous utilisons deux JComboBox, le premier est chargé de données grâce à l'architecture MVC de la bibliothèque swing DefaultComboBoxModel, *le second est chargé de données directement à travers sa méthode* addItem .

La fenêtre comporte :

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

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

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



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

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

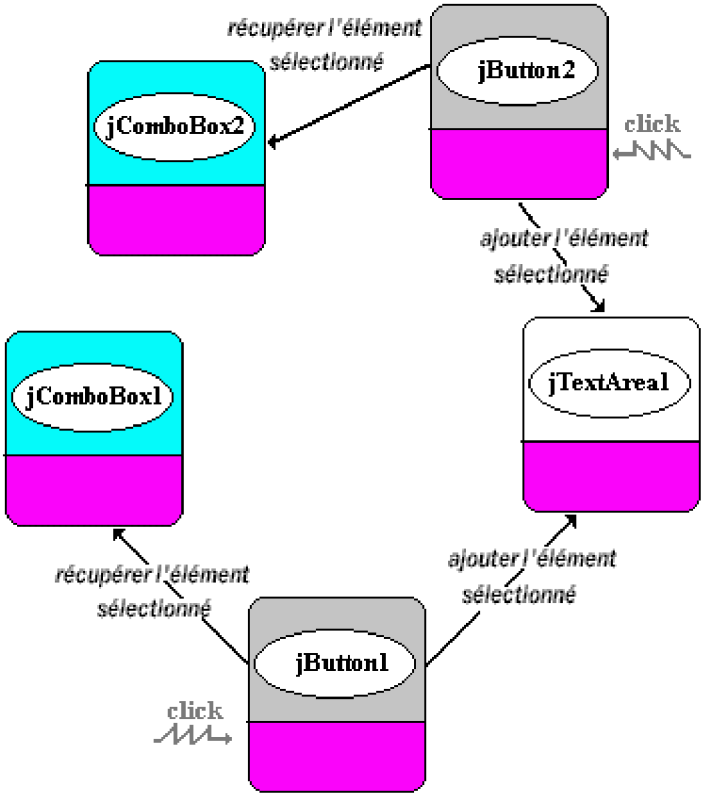
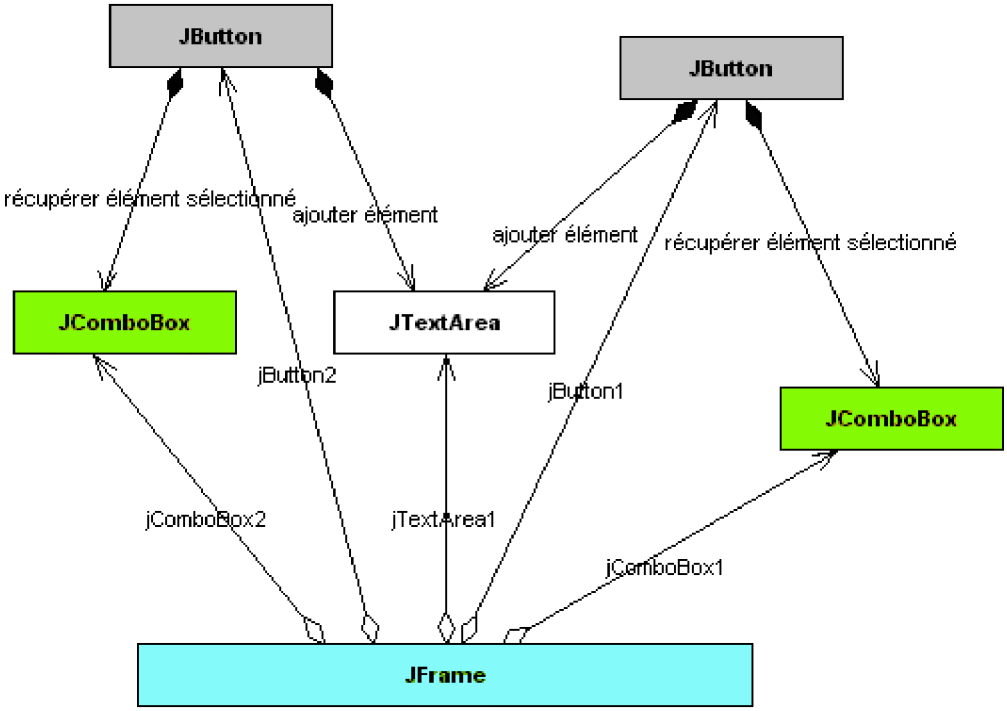
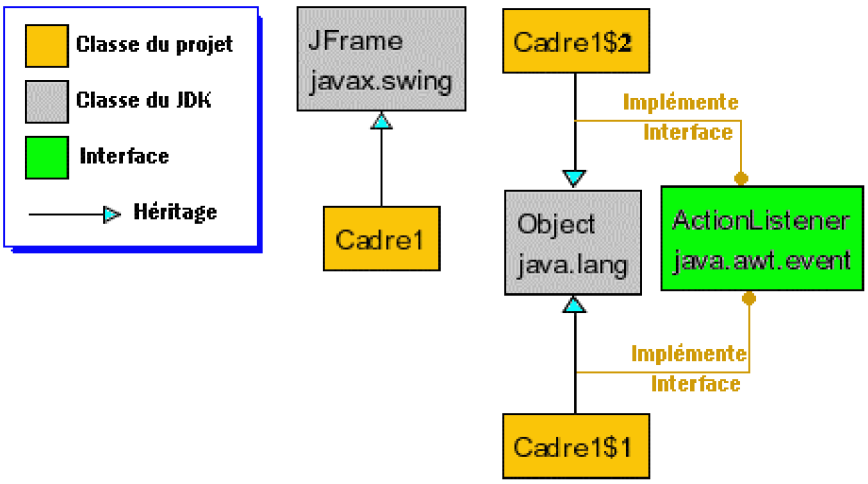


Schéma UML du projet



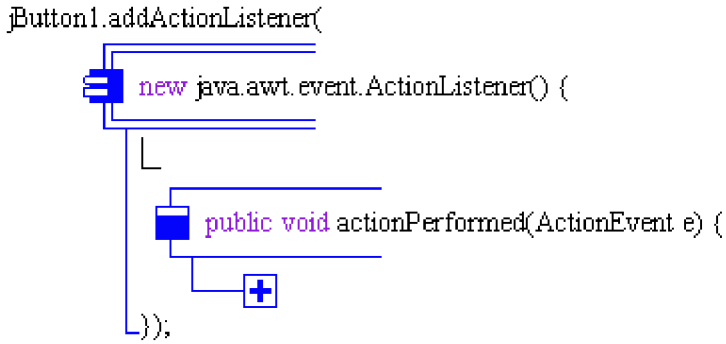
En Java (génération du code source effectuée par JBuilder)

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

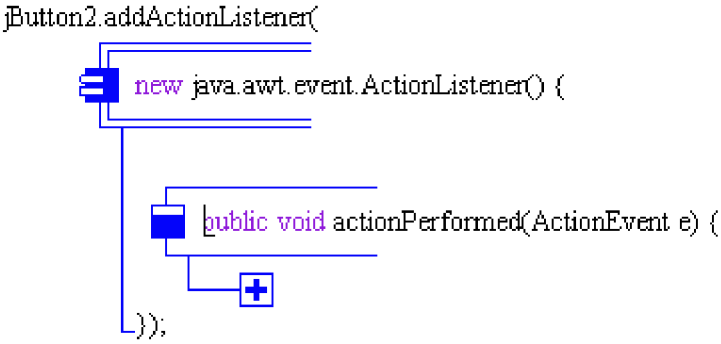


Rappelons que les classes Cadre1$1 et Cadre1$2 sont des classes anonymes créées lors de la déclaration de l'écouteur des boutons jButton1 et jButton2, Java 2 crée donc dynamiquement un objet écouteur interne (dont la référence n'est pas disponible). Ci-dessous les diagrammes jGrasp des classes anonymes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

Cadre1$1:



Cadre1$2:



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

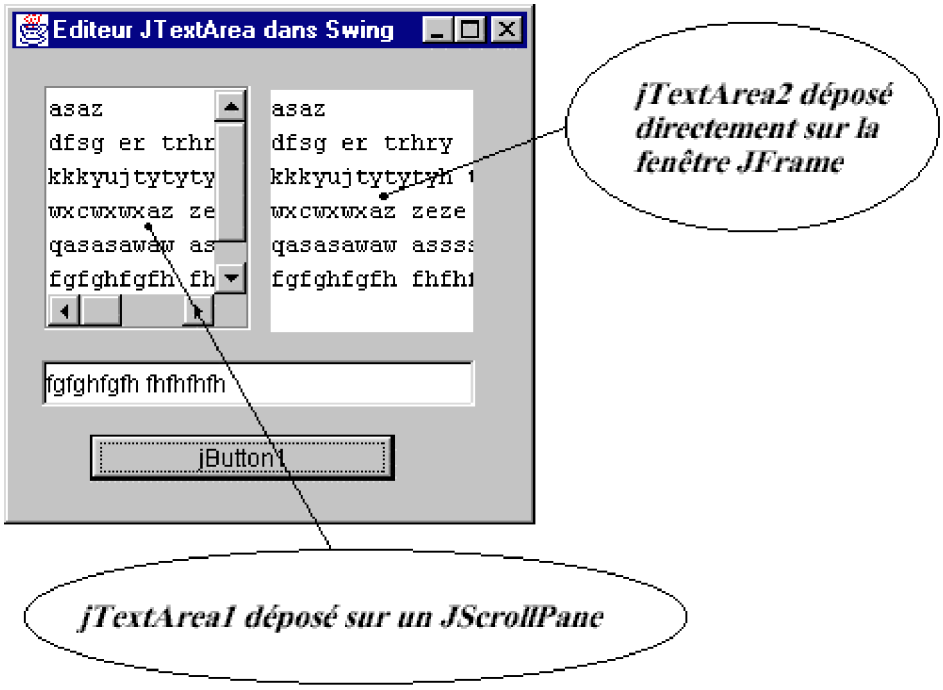
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

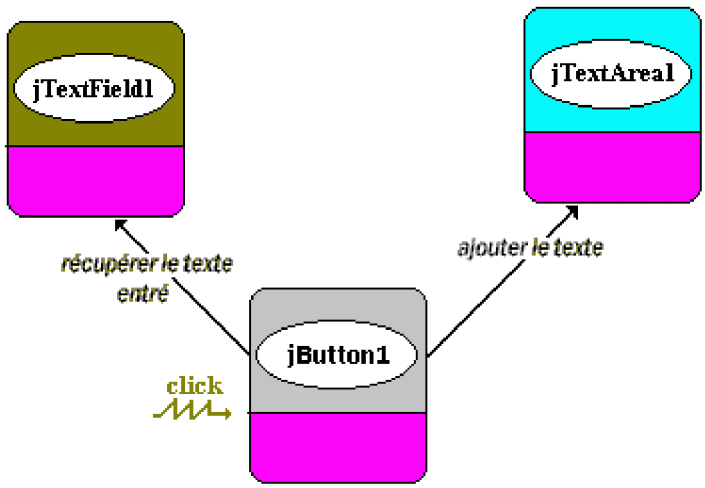
Exemple - JTextArea

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

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



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



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

Gestion des barres de défilement du texte

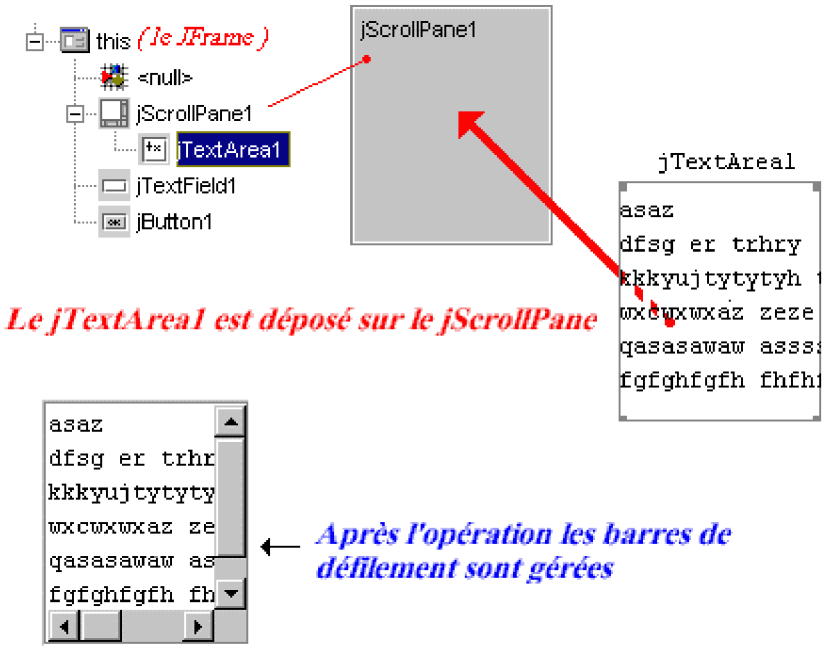
Afin de bien comprendre la gestion des barres de défilement verticales et horizontales, le programme ci-dessous contient deux objets jTextArea1 et jTextArea2 dont le premier est déposé dans un objet conteneur de classe ***JScrollpane*** (classe encapsulant la gestion de barres de défilements), selon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

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

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

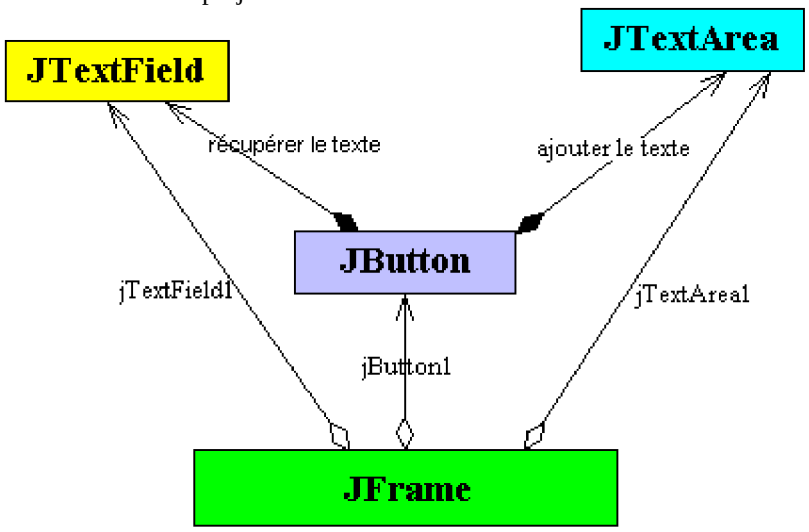
ou bien en utilisant un autre constructeur de JScrollPane :

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

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

Voici en résumé ce qu'il se passe lors de l'exécution de ce code (visualisation avec un RAD permettant d'avoir un explorateur de classes et de conteneur conseillée, ici le traitement est effectué avec JBuilder) :

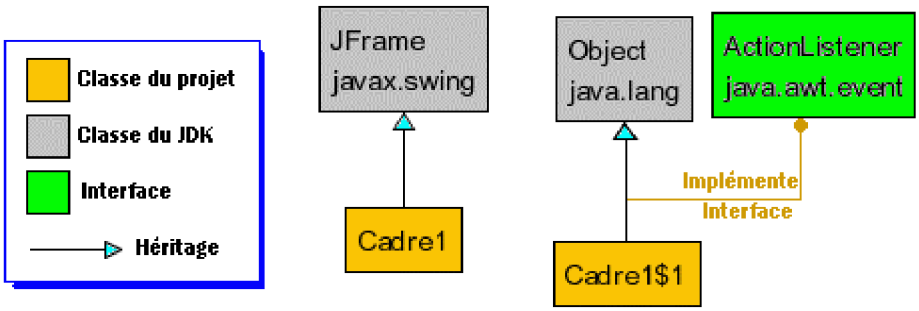


Schémas UML du projet



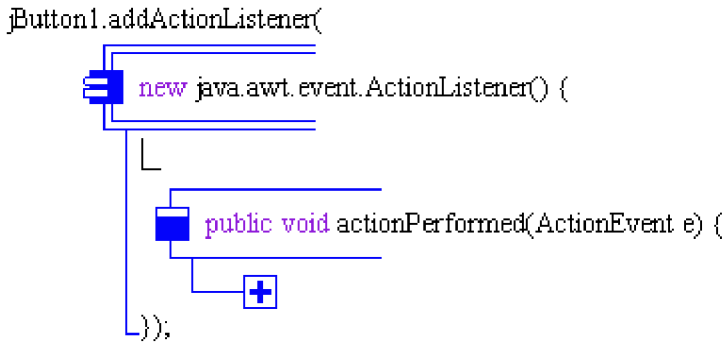
En Java (génération du code source effectuée par JBuilder)

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



Rappelons que la classe Cadre1$1 est une classe anonyme créée lors de la déclaration de l'écouteur du bouton jButton1, Java 2 crée donc dynamiquement un objet écouteur interne (dont la référence n'est pas disponible). Ci-dessous le diagramme jGrasp de la classe anonyme cadre1$1

Cadre1$1:



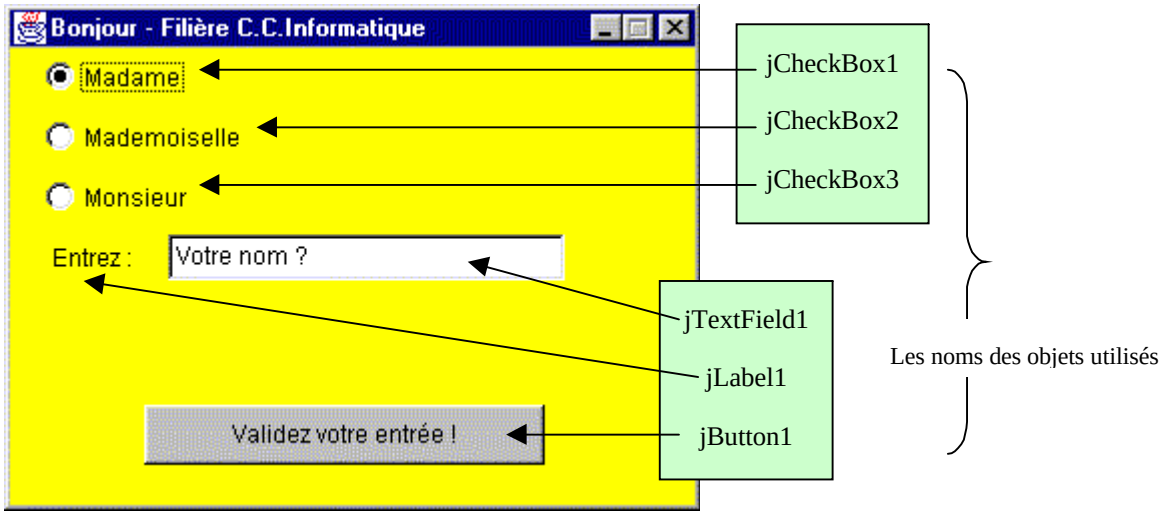
Enfin pour terminer, voici le listing Java/Swing complet de la classe représentant la fenêtre (y compris le deuxième JTextArea de vérification) :

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

public class Cadre1 extends JFrame {
  JTextField jTextField1 = new JTextField();
  JButton jButton1 = new JButton();
  JScrollPane jScrollPane1 = new JScrollPane();
  JTextArea jTextArea2 = new JTextArea();
  JTextArea jTextArea1 = new JTextArea();

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

  //Initialiser le composant
  private void jbInit() throws Exception  {
    this.getContentPane().setLayout(null);
    this.setSize(new Dimension(265, 260));
    this.setTitle("Editeur JTextArea dans Swing");
    jTextField1.setText("jTextField1");
    jTextField1.setBounds(new Rectangle(15, 155, 216, 23));
    jButton1.setText("jButton1");
    jButton1.setBounds(new Rectangle(39, 192, 152, 23));
    jButton1.addActionListener(new java.awt.event.ActionListener() {

      public void actionPerformed(ActionEvent e) {
        jButton1_actionPerformed(e);
      }
    });
    jScrollPane1.setBorder(null);
    jScrollPane1.setBounds(new Rectangle(16, 18, 103, 122));
    jTextArea2.setText("jTextArea2");
    jTextArea2.setBounds(new Rectangle(129, 20, 101, 121));
    jTextArea1.setText("jTextArea1");
    this.getContentPane().add(jScrollPane1, null);
    jScrollPane1.getViewport().add(jTextArea1, null);
    this.getContentPane().add(jTextArea2, null);
    this.getContentPane().add(jTextField1, null);
    this.getContentPane().add(jButton1, null);
  }

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

```
  void jButton1_actionPerformed(ActionEvent e) {
    jTextArea1.append(jTextField1.getText()+"\n");
    jTextArea2.append(jTextField1.getText()+"\n"); // copie pour vérification visuelle
  }
}
```

Saisie interactive de renseignements avec des swing

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



Description événementielle de l'IHM :

- Dans l'IHM au départ le **jButton1** est désactivé, aucun **jCheckBox** n'est coché, le **jTextField1** est vide.
- Dès que l'un des **jCheckBox** est coché, et que le **jTextField1** contient du texte le **jButton1** est activé, dans le cas contraire le **jButton1** est désactivé (dès que le **jTextField1** est vide).
- Un click sur le **jButton1** sauvegarde les informations dans le fichier texte **etudiants.txt**.
- La fiche qui dérive d'une **JFrame** se ferme et arrête l'application sur le click du bouton de fermeture.

```
import java.awt.*; // utilisation des classes du package awt
import java.awt.event.*; // utilisation des classes du package awt
import java.io.*; // utilisation des classes du package io
import javax.swing.*; // utilisation des classes du package swing
import java.util.*; // utilisation des classes du package util pour Enumeration
import javax.swing.text.*;// utilisation pour Document
import javax.swing.event.*;// utilisation pour DocumentEvent

 class ficheSaisie extends JFrame { // la classe ficheSaisie hérite de la classe des fenêtres JFrame
   JButton jbutton1 = new JButton( );// création d'un objet de classe JButton
   JLabel jlabel1 = new JLabel( );// création d'un objet de classe JLabel
   ButtonGroup Group1 = new ButtonGroup ( );// création d'un objet groupe pour AbstractButton et dérivées
```

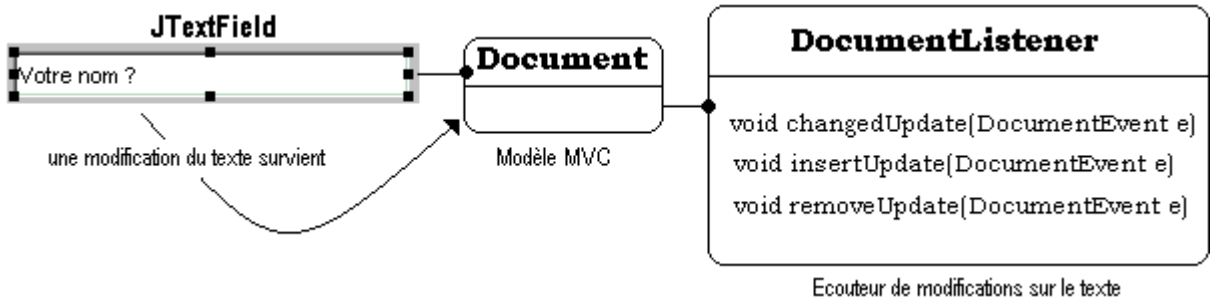
```
   JCheckBox jcheckbox1 = new JCheckBox( );// création d'un objet de classe JCheckbox
   JCheckBox jcheckbox2 = new JCheckBox( );// création d'un objet de classe JCheckbox
   JCheckBox jcheckbox3 = new JCheckBox( );// création d'un objet de classe JCheckbox
   JTextField jtextField1 = new JTextField( );// création d'un objet de classe JTextField
   Container ContentPane;

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


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

//indique quel JCheckBox est coché(réf) ou si aucun n'est coché(null)  :
    private JCheckBox getSelectedjCheckbox(){
      JCheckBox isSelect=null;
      Enumeration checkBenum = Group1.getElements();
      for (int i = 0; i < Group1.getButtonCount(); i++) {
        JCheckBox B =(JCheckBox) checkBenum.nextElement();
        if(B.isSelected())
          isSelect = B;
      }
      return isSelect;
   }
//Active ou désactive le bouton pour sauvegarde :
    private void AutoriserSave(){
      if (jtextField1.getText().length() !=0 &&  this.getSelectedjCheckbox()!=null)
        jbutton1.setEnabled(true);
      else
        jbutton1.setEnabled(false);
   }
//rempli le champ Etatcivil selon le checkBox coché :
    private void StoreEtatcivil(){
      this.AutoriserSave();
      if (this.getSelectedjCheckbox()!=null)
        this.EtatCivil=this.getSelectedjCheckbox().getLabel();
      else
        this.EtatCivil="";
   }
//sauvegarde les infos étudiants dans le fichier :
    public void ecrireEnreg(String record) {
      try {
        fluxout.write(record);//écrit les infos
        fluxout.newLine( ); //écrit le eoln
      }
         catch (IOException err) {
           System.out.println( "Erreur : " + err );      }
   }

//Initialiser la fenêtre :
    private void Initialiser( ) {  //Création et positionnement de tous les composants
      ContentPane = this.getContentPane( ) ; //Référencement du fond de dépôt des composants
      this.setResizable(false); // la fenêtre ne peut pas être retaillée par l'utilisateur
      ContentPane.setLayout(null); // pas de Layout, nous positionnons les composants nous-mêmes
      ContentPane.setBackground(Color.yellow); // couleur du fond de la fenêtre
      this.setSize(348, 253); // width et height de la fenêtre
      this.setTitle("Bonjour - Filière C.C.Informatique"); // titre de la fenêtre
```

```
this.setForeground(Color.black); // couleur de premier plan de la fenêtre
jbutton1.setBounds(70, 160, 200, 30); // positionnement du bouton
jbutton1.setText("Validez votre entrée !"); // titre du bouton
jbutton1.setEnabled(false); // bouton désactivé
jlabel1.setBounds(24, 115, 50, 23); // positionnement de l'étiquette
jlabel1.setText("Entrez :"); // titre de l'étiquette
jcheckbox1.setBounds(20, 25, 88, 23); // positionnement du JCheckBox
Group1.add(jcheckbox1); // ce JCheckBox est mis dans le groupe Group1
jcheckbox1.setText("Madame");// titre du JCheckBox
jcheckbox2.setBounds(20, 55, 108, 23);// positionnement du JCheckBox
Group1.add(jcheckbox2); // ce JCheckBox est mis dans le groupe Group1
jcheckbox2.setText("Mademoiselle");// titre du JCheckBox
jcheckbox3.setBounds(20, 85, 88, 23);// positionnement du JCheckBox
Group1.add(jcheckbox3); // ce JCheckBox est mis dans le groupe Group1
jcheckbox3.setText("Monsieur");// titre du JCheckBox
jcheckbox1.setBackground(Color.yellow);//couleur du fond du jcheckbox1
jcheckbox2.setBackground(Color.yellow);//couleur du fond du jcheckbox2
jcheckbox3.setBackground(Color.yellow);//couleur du fond du jcheckbox3
jtextField1.setBackground(Color.white);// couleur du fond de l'éditeur mono ligne
jtextField1.setBounds(82, 115, 198, 23);// positionnement de l'éditeur mono ligne
jtextField1.setText("Votre nom ?");// texte de départ de l'éditeur mono ligne
ContentPane.add(jcheckbox1);// ajout dans la fenêtre du JCheckBox
ContentPane.add(jcheckbox2);// ajout dans le ContentPane  de la fenêtre du JCheckBox
ContentPane.add(jcheckbox3);// ajout dans le ContentPane  de la fenêtre du JCheckBox
ContentPane.add(jbutton1);// ajout dans le ContentPane  de la fenêtre du bouton
ContentPane.add(jtextField1);// ajout dans le ContentPane  de la fenêtre de l'éditeur mono ligne
ContentPane.add(jlabel1);// ajout dans le ContentPane  de la fenêtre de l'étiquette
EtatCivil ="";// pas encore de valeur
Document doc = jtextField1.getDocument();// partie Modele MVC du JTextField
try{
  fluxwrite = new FileWriter("etudiants.txt",true);// création du fichier(en mode ajout)
  fluxout = new BufferedWriter(fluxwrite); //tampon de ligne associé
}
   catch(IOException err){ System.out.println( "Problème dans l'ouverture du fichier ");}
//--> événements et écouteurs :
this.addWindowListener(
     new WindowAdapter(){
       public void windowClosing(WindowEvent e){
         try{
           fluxout.close( ); //le fichier est fermé et le tampon vidé
         }
           catch(IOException err){ System.out.println( "Impossible de fermer le fichier ");}
         System.exit(100);
       }
     });
doc.addDocumentListener(//on crée un écouteur anonymepour le  Modele (qui fait aussi le controle)
     new javax.swing.event.DocumentListener() {
      //-- l'interface DocumentListener a 3 méthodes qu'il faut implémenter :
       public void changedUpdate(DocumentEvent e) {}

       public void removeUpdate(DocumentEvent e) {//une suppression est un changement de texte
         textValueChanged(e);// appel au gestionnaire de changement de texte
       }

       public void insertUpdate(DocumentEvent e) {//une insertion est un changement de texte
         textValueChanged(e);// appel au gestionnaire de changement de texte
       }
     });
jcheckbox1.addItemListener(
     new ItemListener(){
```

```
            public void itemStateChanged(ItemEvent e){
              StoreEtatcivil( );
            }
          });
      jcheckbox2.addItemListener(
          new ItemListener(){
            public void itemStateChanged(ItemEvent e){
              StoreEtatcivil();
            }
          });
      jcheckbox3.addItemListener(
          new ItemListener(){
            public void itemStateChanged(ItemEvent e){
              StoreEtatcivil();
            }
          });
      jbutton1.addActionListener(
          new java.awt.event.ActionListener() {
            public void actionPerformed(ActionEvent e) {
              ecrireEnreg(EtatCivil+":"+jtextField1.getText());
            }
          });
    }
    //Gestionnaire du changement de texte dans un document
     private void textValueChanged(DocumentEvent e) {
      AutoriserSave();
    }
  }
```

Remarques: (comparez ce code au même exercice construit en Awt et notez les différences)

- Un JTextField n'est pas comme un TextField directement sensible au changement de son texte, c'est son modèle MVC du type Document qui assure la gestion et la réaction à la survenue de modifications du texte en recensant des écouteurs de classe héritant de l'interface DocumentListener :



Un **ButtonGroup** ne possède pas comme un **CheckboxGroup** Awt , de méthode **getSelectedjCheckbox** permettant de connaître le bouton du groupe qui est coché. Nous avons construit une telle méthode **getSelectedjCheckbox** qui renvoie la référence d'un **JCheckBox** semblablement à la méthode des Awt :

```
  //-- indique quel JCheckBox est coché(réf) ou si aucun n'est coché(null)  :
   private JCheckBox getSelectedjCheckbox(){
    JCheckBox isSelect=null;
    Enumeration checkBenum = Group1.getElements();
    for (int i = 0; i < Group1.getButtonCount(); i++) {
      JCheckBox B =(JCheckBox) checkBenum.nextElement();
      if(B.isSelected())
        isSelect = B;
    }
    return isSelect;
  }
```

Les applets Java : applications internet

La page HTML minimale

Jusqu'à présent tous les programmes Java qui ont été écrits dans les autres chapitres sont des applications autonomes pouvant fonctionner directement sur une plateforme Windows, Unix, Linux, MacOs...

Une des raisons initiales, du succès de Java peut être la raison majeure, réside dans la capacité de ce langage à créer des applications directement exécutables dans un navigateur contenant une machine virtuelle java. Ces applications doivent être insérées dans le code interne d'une page HTML (**H**yper**T**ext **M**arkup **L**anguage) entre deux balises spécifiques.

Ces applications insérées dans une page HTML, peuvent être exécutées en local sur la machine hôte, le navigateur jouant le rôle d'environnement d'exécution. Elles peuvent aussi être exécutées en local par votre navigateur pendant une connexion à internet par téléchargement de l'application en même temps que le code HTML de la page se charge. Ces applications Java non autonomes sont dénommées des **applets**.

Applet = **Appl**ication Intern**et** écrite en Java intégrée à une page HTML ne pouvant être qu'exécutée par un navigateur et s'affichant dans la page HTML.

Une page HTML est un fichier texte contenant des descriptions de la mise en page du texte et des images. Ces descriptions sont destinées au navigateur afin qu'il assure l'affichage de la page, elles s'effectuent par l'intermédiare de balises (parenthèses de description contextuelles). Un fichier HTML commence toujours par la balise <HTML> et termine par la balise </HTML>.

Une page HTML minimale contient deux sections :

- l'en-tête de la page balisée par <HEAD> ...</HEAD>
- le corps de la page balisé par <BODY> ... </BODY>

Voici un fichier texte minimal pour une page HTML :

```
<HTML>
    <HEAD>

    </HEAD>
    <BODY>
    </BODY>
```

</HTML>

Une applet Java est invoquée grâce à deux balises spéciales insérées au sein du corps de la page HTML :

<**APPLET CODE** =.....>

</**APPLET**>

Nous reverrons plus loin la signification des paramètres internes à la balise <APPLET>.

Terminons cette introduction en signalant qu'une applet, pour des raisons de sécurité, est soumise à certaines restrictions notamment en matière d'accès aux disques durs de la machine hôte.

La classe java.applet.Applet

Cette classe joue un rôle semblable à celui de la classe Frame dans le cas d'une application. Une Applet est un Panel spécial qui se dessinera sur le fond de la page HTML dans laquelle il est inséré.

Voici la hiérarchie des classes dont Applet dérive :

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--java.awt.Panel
                    |
                    +--java.applet.Applet
```

Etant donné l'importance de la classe, nous livrons ci-dessous, la documentation du JDK de la classe Applet.

Minimum requis pour une applet

Si vous voulez écrire des applets, il faut posséder un navigateur pour exécuter les applet et un éditeur de texte permettant d'écrire le code source Java.

Vous pouvez aussi utiliser un environnement de développement Java

Tout environnement de développement java doit contenir un moyen de visionner le résultat de la programmation de votre applet, cet environnement fera appel à une visionneuse d'applet (dénommée AppletViewer dans le JDK).

Votre applet doit hériter de la classe Applet

public class AppletExemple **extends** Applet { .....
}
Comme toutes les classes Java exécutables, le nom du fichier doit correspondre très exactement au nom de la classe avec comme suffixe java (ici : **AppletExemple.java**)

Constructor Summary

Applet()	

Method Summary

void	**destroy**() Called by the browser or applet viewer to inform this applet that it is being reclaimed and that it should destroy any resources that it has allocated.
AppletContext	**getAppletContext**() Determines this applet's context, which allows the applet to query and affect the environment in which it runs.
String	**getAppletInfo**() Returns information about this applet.
AudioClip	**getAudioClip**(URL url) Returns the AudioClip object specified by the URL argument.
AudioClip	**getAudioClip**(URL url, String name) Returns the AudioClip object specified by the URL and name arguments.
URL	**getCodeBase**() Gets the base URL.
URL	**getDocumentBase**() Gets the document URL.
Image	**getImage**(URL url) Returns an Image object that can then be painted on the screen.
Image	**getImage**(URL url, String name) Returns an Image object that can then be painted on the screen.

Locale	**getLocale**() Gets the Locale for the applet, if it has been set.
String	**getParameter**(String name) Returns the value of the named parameter in the HTML tag.
String[][]	**getParameterInfo**() Returns information about the parameters than are understood by this applet.
void	**init**() Called by the browser or applet viewer to inform this applet that it has been loaded into the system.
boolean	**isActive**() Determines if this applet is active.
static AudioClip	**newAudioClip**(URL url) Get an audio clip from the given URL
void	**play**(URL url) Plays the audio clip at the specified absolute URL.
void	**play**(URL url, String name) Plays the audio clip given the URL and a specifier that is relative to it.
void	**resize**(Dimension d) Requests that this applet be resized.
void	**resize**(int width, int height) Requests that this applet be resized.
void	**setStub**(AppletStub stub) Sets this applet's stub.
void	**showStatus**(String msg) Requests that the argument string be displayed in the "status window".
void	**start**() Called by the browser or applet viewer to inform this applet that it should start its execution.
void	**stop**() Called by the browser or applet viewer to inform this applet that it should stop its execution.

Fonctionnement d'une applet

Nous allons examiner quelques unes des 23 méthodes de la classe Applet, essentielles à la construction et à l'utilisation d'une applet.

La méthode init( )

Lorsqu'une applet s'exécute la méthode principale qui s'appelait **main( )** dans le cas d'une application Java se dénomme ici **init( )**. Il nous appartient donc de **surcharger dynamiquement**

(redéfinir) la méthode **init** de la classe Applet afin que notre applet fonctionne selon nos attentes.

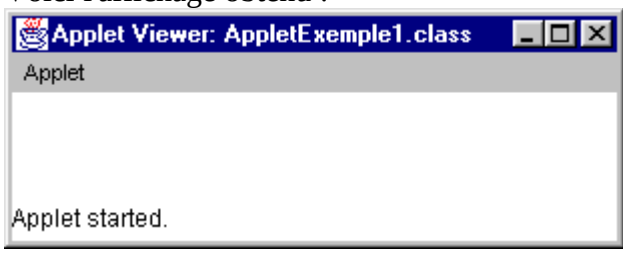
La méthode **init( )** est appelée **une seule fois**, lors du chargement de l'applet avant que celui-ci ne soit affiché.

Exemple d'applet vide :

Code Java

```
import java.applet.*;
public class AppletExemple1 extends Applet  {


}
```

Voici l'affichage obtenu :



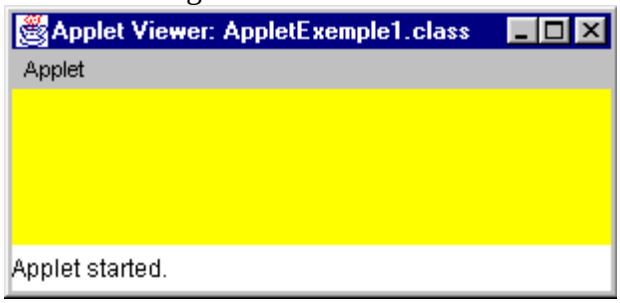
Soit à colorier le fond de l'applet, nous allons programmer le changement du fond de l'objet applet à partir de la méthode **init** de l'applet (avant même qu'il ne s'affiche).

```
public void init( ) {
 // avant le démarrage de l'affichage de l'applet
 this.setBackground(Color.yellow); // équivalent à : setBackground(Color.yellow);
}
```

Code Java

```
import java.applet.*;
public class AppletExemple1 extends Applet
{  public void init( ) {
     this.setBackground(Color.yellow);
   }
}
```

Voici l'affichage obtenu :



La méthode start( )

La méthode **start( )** est appelée **à chaque fois**, soit :

- après la méthode init
- après chaque modification de la taille de l'applet (minimisation,...).

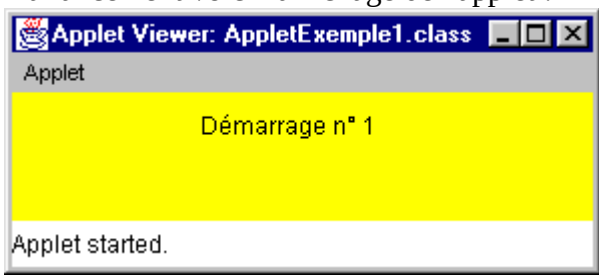
Si nous voulons que notre applet effectue une action spécifique au redémarrage, il suffit de **surcharger dynamiquement (redéfinir)** la méthode **start( )**.

Soit à compter et afficher dans une Label, le nombre d'appels à la méthode start :

Code Java

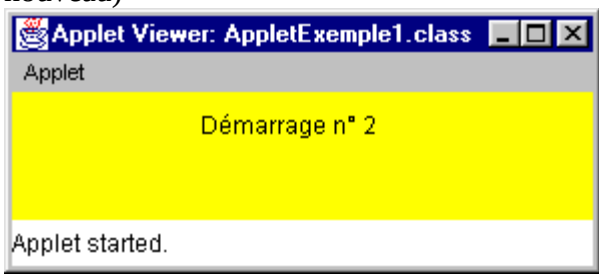
```
import java.applet.*;
public class AppletExemple1 extends Applet
{   Label etiq = new Label("Démarrage n&deg; ");
    int nbrDemarr = 0 ;
    public void init( ) {
    this.setBackground(Color.yellow);
    this.add(etiq);
   }
   public void start( ) {
     nbrDemarr++;
     etiq.setText( "Démarrage n&deg; "+String.valueOf ( nbrDemarr ) );
   }
}
```

Au lancement voici l'affichage de l'applet :



On minimise l'applet dans la bare des tâches :

On restaure l'applet à partir de la barre des tâches : (**start** est appelée automatiquemen à nouveau)



etc....

La méthode paint( )

La méthode **paint(Graphics x)** est appelée **à chaque fois**, soit :

- après que l'applet a été masquée, déplacée, retaillée,...
- à chaque réaffichage de l'applet (après minimisation,...).

Cette méthode est chargée de l'affichage de tout ce qui est graphique, vous mettrez dans le corps de cette méthode votre code d'affichage de vos éléments graphiques afin qu'ils soient redessinés systématiquement.

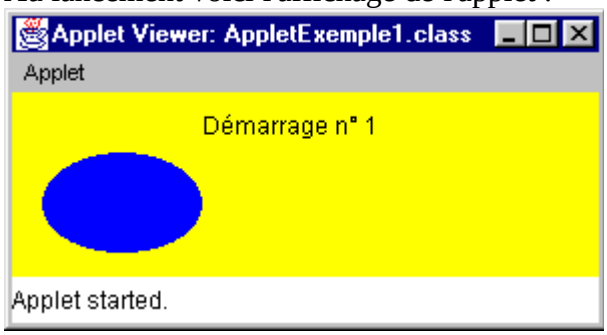
Si nous voulons que notre applet redessine nos graphiques, il suffit de **surcharger dynamiquement (redéfinir)** la méthode **paint(Graphics x)**.

Soit à dessiner dans l'applet précédent une ellipse peinte en bleu, on rajoute le code dans la méthode paint :

Code Java

```
import java.applet.*;
public class AppletExemple1 extends Applet
{   Label etiq = new Label("Démarrage n&deg; ");
     int nbrDemarr = 0 ;
     public void init( ) {
     this.setBackground(Color.yellow);
     this.add(etiq);
    }
    public void start( ) {
      nbrDemarr++;
      etiq.setText( "Démarrage n&deg; "+String.valueOf ( nbrDemarr ) );
    }
   public void paint (Graphics x) {
     this.setForeground(Color.blue);
     x.fillOval(15,30,80,50);
   }
}
```

Au lancement voici l'affichage de l'applet :



Les méthodes stop( ) et destroy( )

La méthode **stop( )** est appelée **à chaque fois**, soit :

- que l'applet a été masquée dans la page du navigateur (défilement vertical ou horizontal dans le navigateur), déplacée, retaillée,...
- lors de l'abandon et du changement de la page dans le navigateur.

Cette méthode arrête toutes les actions en cours de l'applet.

Si nous voulons que notre applet effectue des actions spécifiques lors de son arrêt (comme désactiver ou endormir des Threads, envoyer des messages...), il suffit de **surcharger dynamiquement (redéfinir)** la méthode **stop( )**.

La méthode **destroy( )** est appelée **à chaque fois**, soit :

- que l'utilisateur charge une nouvelle page dans le navigateur
- lors de la fermeture du navigateur

Pour mémoire la méthode **destroy( )** est invoquée pour libérer toutes les ressources que l'applet utilisait, normalement la machine virtuelle Java se charge de récupérer automatiquement la mémoire.

Si nous voulons que notre applet effectue des actions spécifiques lors de son arrêt (comme terminer définitivement des Threads), il suffit de **surcharger dynamiquement (redéfinir)** la méthode **destroy( )**.

Une applet dans une page HTML

Le code d'appel d'une applet

Nous avons déjà indiqué au premier paragraphe de ce chapitre que le code d'appel d'une applet est intégré au texte source de la page dans laquelle l'applet va être affichée. Ce sont les balises <APPLET CODE =.....> et </APPLET> qui précisent les modalités de fonctionnement de l'applet. En outre, l'applet s'affichera dans la page exactement à l'emplacement de la balise dans le code HTML. Donc si l'on veut déplacer la position d'une applet dans une page HTML, il suffit de déplacer le code compris entre les deux balises <APPLET CODE =.....> et </APPLET>.

Voici une page HTML dont le titre est "Simple Applet " et ne contenant qu'une applet :

```
<HTML>
  <HEAD>

    <TITLE> Simple Applet </TITLE>

  </HEAD>

  <BODY>
    <APPLET CODE="AppletSimple.class" WIDTH=200 HEIGHT=100>
    </APPLET>
  </BODY>

</HTML>
```

Il y a donc des paramètres obligatoires à transmettre à une applet, ce sont les paramètres CODE, WIDTH et HEIGHT :

Paramètre	signification
CODE	le nomp du fichier contenant la classe applet à afficher.
WIDTH	la largeur de l'applet au démarrage dans la page HTML (en pixels)
HEIGHT	la hauteur de l'applet au démarrage dans la page HTML (en pixels)

A côté de ces paramètres obligatoires existent des paramètres facultatifs relatifs au positionnement et à l'agencement de l'applet à l'intérieur de la page HTML (align, alt, hspace, vspace, codebase, name).

La balise interne PARAM

Il est possible, à l'intérieur du corps de texte entre les balises <APPLET CODE =.....> et

</APPLET> , d'introduire un marqueur interne de paramètre <PARAM .....>, que le fichier HTML transmet à l'applet. Ce marqueur possède obligatoirement deux attributs,soit **name** le nom du paramètre et **value** la valeur du paramètre sous forme d'une chaîne de caractères :

<PARAM name = "NomduParam1" value = "une valeur quelconque">

La classe **applet** dispose d'une méthode permettant de récupérer la valeur associée à un paramètre dont le nom est connu (il doit être strictement le même que celui qui se trouve dans le texte HTML !), cette méthode se dénomme getParameter.

Ci-dessous le même applet que précédemment à qui l'on passe un paramètre de nom **par1** dans le marqueur PARAM.

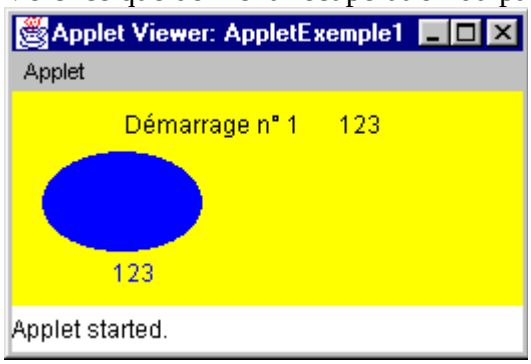
Code Java de l'applet

```
import java.applet.*;
public class AppletExemple1 extends Applet
{   Label etiq = new Label("Démarrage n&deg; ");
    int nbrDemarr = 0 ;
    String valparam ;
    public void init( ) {
    this.setBackground(Color.yellow);
    this.add(etiq);
    valparam = this.getParameter("par1");
    this.add(new Label(valparam));
   }
   public void start( ) {
     nbrDemarr++;
     etiq.setText( "Démarrage n&deg; "+String.valueOf ( nbrDemarr ) );
   }
  public void paint (Graphics x) {
    this.setForeground(Color.blue);
    x.fillOval(15,30,80,50);
    x.drawString(valparam,50,95);
  }
}
```

Code HTML d'affichage de l'applet

```
<HTML>
<HEAD>
<TITLE> Filière CCI </TITLE>
</HEAD>
<BODY>

<APPLET CODE="AppletExemple1" WIDTH=250 HEIGHT=150>
<PARAM  NAME = "par1"  VALUE = "123">
</APPLET>

</BODY>
</HTML>
```

Voici ce que donne la récupération du paramètre



Nous ne sommes pas limités au passage d'un seul paramètre, il suffit de mettre autant de marqueur PARAM avec des noms de paramètres différents que l'on veut.

Soit l'exemple précédent repris avec deux paramètres : **par1** et **par2**

Code Java de l'applet

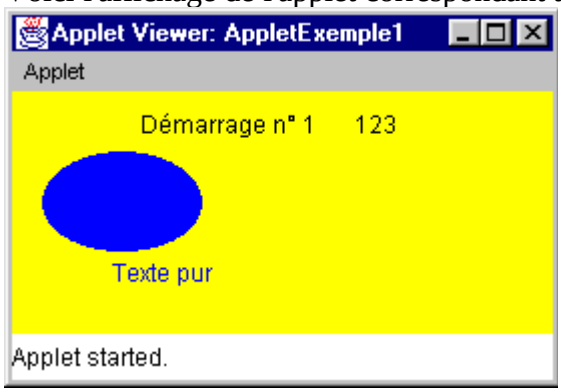
```
import java.applet.*;
public class AppletExemple1 extends Applet
{   Label etiq = new Label("Démarrage n&deg; ");
     int nbrDemarr = 0 ;
     String valparam ;
     public void init( ) {
     this.setBackground(Color.yellow);
     this.add(etiq);
     valparam = this.getParameter("par1");
     this.add(new Label(valparam));
    }
    public void start( ) {
      nbrDemarr++;
      etiq.setText( "Démarrage n&deg; "+String.valueOf ( nbrDemarr ) );
    }
   public void paint (Graphics x) {
     this.setForeground(Color.blue);
     x.fillOval(15,30,80,50);
     x.drawString(getParameter("par2"),50,95);
   }
}
```

La valeur de **par1** reste la même soit "123", celle de **par2** est la phrase "Texte pur"

Code HTML d'affichage de l'applet

```
<HTML>
<HEAD>  <TITLE> Filière CCI </TITLE> </HEAD>
<BODY>

    <APPLET CODE="AppletExemple1" WIDTH=250 HEIGHT=150>
    <PARAM  NAME = "par1"  VALUE = "123">
    <PARAM  NAME = "par2" VALUE = "Texte pur">
    </APPLET>

</BODY>
</HTML>
```

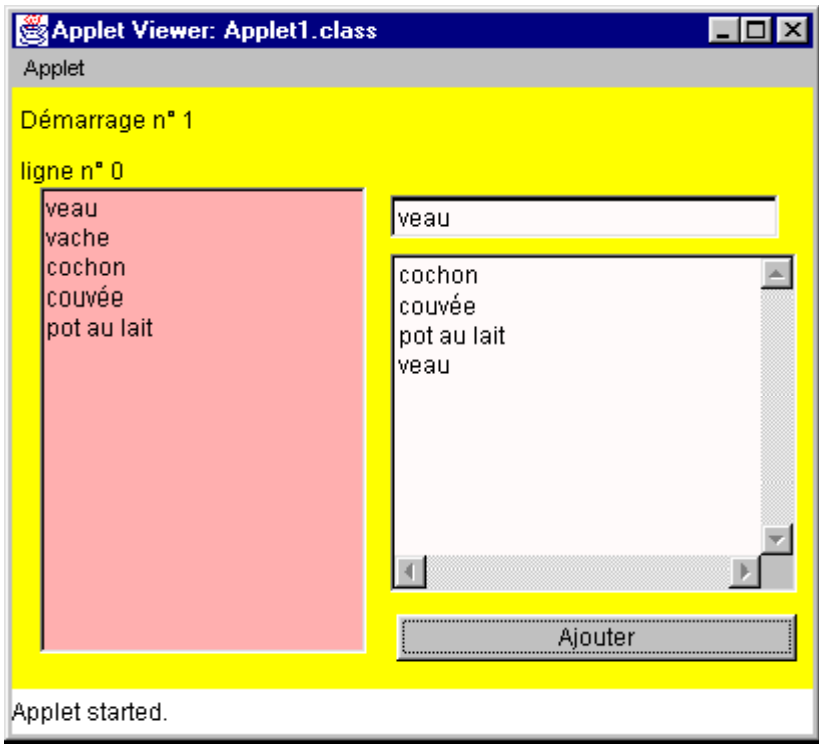
Voici l'affichage de l'applet correspondant à cette combinaison :



AWT dans les applets : exemples

Comme une applet est une classe héritant des Panel, c'est donc un conteneur de composants et donc tout ce que nous avons appris à utiliser dans les classes du package AWT, reste valide pour une applet.

Exemple - 1 : Interface à quatre composants



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

/* interception du double click de souris par actionPerformed :
- c'est le double click sur un élément de la liste qui déclenche l'action.
list1.addActionListener(new java.awt.event.ActionListener() {

     public void actionPerformed(ActionEvent e) {
       list1_actionPerformed(e);
     }
   });
- même gestion actionPerformed pour le click sur le bouton :
    c'est le click sur le bouton qui déclenche l'action.
*/
 public class Applet0 extends Applet {
      boolean isStandalone = false;
      Label etiq = new Label("Démarrage n&deg;");
      Label numero = new Label("ligne n&deg;");
      int nbrDemarr = 0;
      List list1 = new List();
      TextField textField1 = new TextField();
      TextArea textArea1 = new TextArea();
      Button button1 = new Button();

      // Initialiser l'applet
      public void init( )  {
        this.setSize(new Dimension(400,300));
        this.setLayout(null);
        this.setBackground(Color.yellow);
        etiq.setBounds(new Rectangle(4, 4, 163, 23));
        numero.setBounds(new Rectangle(4, 27, 280, 23));
        list1.setBackground(Color.pink);
        list1.setBounds(new Rectangle(14, 50, 163, 233));
        list1.add("veau");
        list1.add("vache");
        list1.add("cochon");
        list1.add("couvée");
        list1.add("pot au lait");
        list1.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(ActionEvent e) {
              list1_actionPerformed(e);
           }
         });
        button1.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(ActionEvent e) {
              button1_actionPerformed(e);
           }
         });
        textField1.setBackground(Color.pink);
        textField1.setBounds(new Rectangle(189, 54, 194, 21));
        textArea1.setBackground(Color.pink);
        textArea1.setBounds(new Rectangle(189, 83, 203, 169));
        button1.setBounds(new Rectangle(192, 263, 200, 23));
        button1.setLabel("Ajouter");
        this.add(list1, null);
        this.add(etiq, null);
        this.add(numero, null);
```

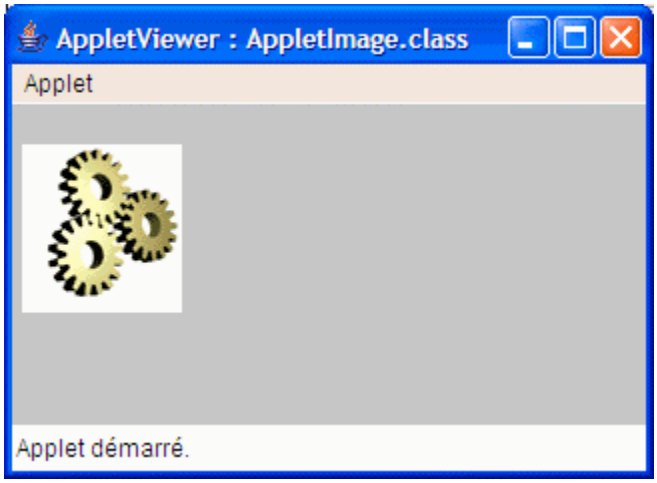
```
        this.add(textField1, null);
        this.add(textArea1, null);
        this.add(button1, null);
        this.setForeground(Color.black);
      }

      // Démarrage de l'applet
     public void start( ) {
       nbrDemarr++;
       etiq.setText("Démarrage n&deg; "+String.valueOf(nbrDemarr));
      }

     // Méthodes d'événement redéfinies
     void list1_actionPerformed(ActionEvent e) {
       int rang = list1.getSelectedIndex( );
       numero.setText("ligne n&deg; "+String.valueOf(rang));
       if (rang>=0) {
         textField1.setText(list1.getSelectedItem( ));
        list1.deselect(list1.getSelectedIndex( ));
        }
       else textField1.setText("rien de sélectionné!");
       }

     void button1_actionPerformed(ActionEvent e) {
         textArea1.append(textField1.getText( )+"\n");
      }
    }
```

On pouvait aussi intercepter les événements au bas niveau directement sur le click de souris, en utilisant toujours des classes anonymes, dérivant cette fois de MouseAdapter et en redéfinissant la méthode mouseClicked :

```
list1.addMouseListener(new MouseAdapter() {
                public void mouseClicked(MouseEvent e) {
                list1_mouseClicked(e);
                }
          });

button1.addMouseListener(new MouseAdapter() {
              public void mouseClicked(MouseEvent e) {
                button1_mouseClicked(e);
                }
         });

void list1_mouseClicked(MouseEvent e) {
// getClickCount indique combien de click ont été efectués sur l'objet (double click = 2 clicks)
     if (e.getClickCount( ) == 2) {
      int rang = list1.getSelectedIndex();
      numero.setText("ligne n&deg; "+String.valueOf(rang));
      if (rang>=0) {
          textField1.setText(list1.getSelectedItem());
       list1.deselect(list1.getSelectedIndex());
         }
      else textField1.setText("rien de sélectionné!");
      }
 }

void button1_mouseClicked(MouseEvent e) {
// remarque : un click = un pressed + un released
```

```
        textArea1.append(textField1.getText()+"\n");
}
```

Exemple - 2 : Afficher une image (gif animé)




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

   public class AppletImage extends Applet {
     Image uneImage;// une référence d'objet de type Image

      // Initialiser l'applet
      public void init( )  {
        this.setBackground(Color.lightGray); //couleur du fond
        uneImage = getImage(this.getCodeBase( ),"rouage.gif");
        this.setSize(200,160);
      }
      // Dessinement de l'applet
      public void paint(Graphics g ) {
       if(uneImage != null)
       g.drawImage(uneImage,5,20,this);
      }
   }
```

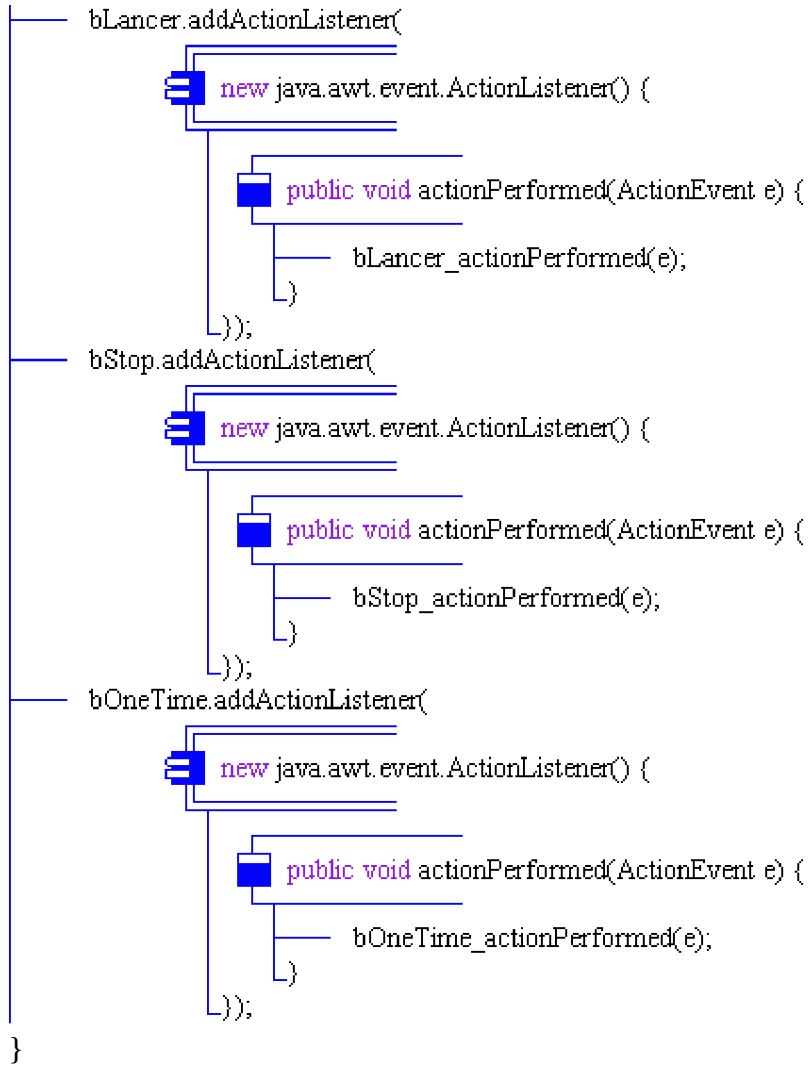
Exemple - 3 : Jouer un morceau de musique

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

```
public class AppletMusic1 extends Applet{

  AudioClip music = null;
  Button bLancer = new Button("Ecouter la musique en continu");
  Button bStop = new Button("Stopper la musique");
  Button bOneTime = new Button("Ecouter la musique une seule fois");

  public void init(){
    +

  void bLancer_actionPerformed(ActionEvent e){
    // musique en continu
    music.loop();
  }

  void bStop_actionPerformed(ActionEvent e){
    //Arrêter la musique
    music.stop();
  }

  void bOneTime_actionPerformed(ActionEvent e){
    //musique une seule fois
    music.play();
  }

  public void destroy(){
    music.stop();
  }
}
```

Contenu de la méthode init ( ) :

```
public void init(){
    music = getAudioClip(getCodeBase(),"20TH_CEN.WAV");
    this.setSize(new Dimension(400,100));
    add(bLancer);
    add(bStop);
    add(bOneTime);
```

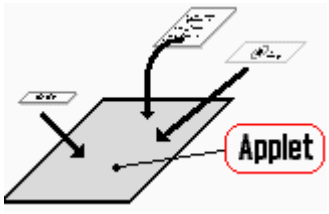

bLancer.addActionListener(
new java.awt.event.ActionListener() {
public void actionPerformed(ActionEvent e) {
bLancer_actionPerformed(e);
}
});
bStop.addActionListener(
new java.awt.event.ActionListener() {
public void actionPerformed(ActionEvent e) {
bStop_actionPerformed(e);
}
});
bOneTime.addActionListener(
new java.awt.event.ActionListener() {
public void actionPerformed(ActionEvent e) {
bOneTime_actionPerformed(e);
}
});
}

Afficher des composants, redessiner une Applet Awt - Swing

Objectif : Comparatif de programmation d'affichage et de redessinement de composants visuels sur une applet Awt et sur une applet Swing. On souhaite redessiner les composants visuels déposés sur une applet lorsqu'ils ont été masqués (par une autre fenêtre) et qu'il nous faut les réafficher.

1. Avec la bibliothèque Awt

En Awt on dépose directement des composants visuels avec la méthode add sur une applet de classe **Applet**. En fait le dessin du composant déposé s'effectue sur l'objet **Canvas** de l'applet (**Canvas** est une classe dérivée de Component destinée à traiter des dessins) :



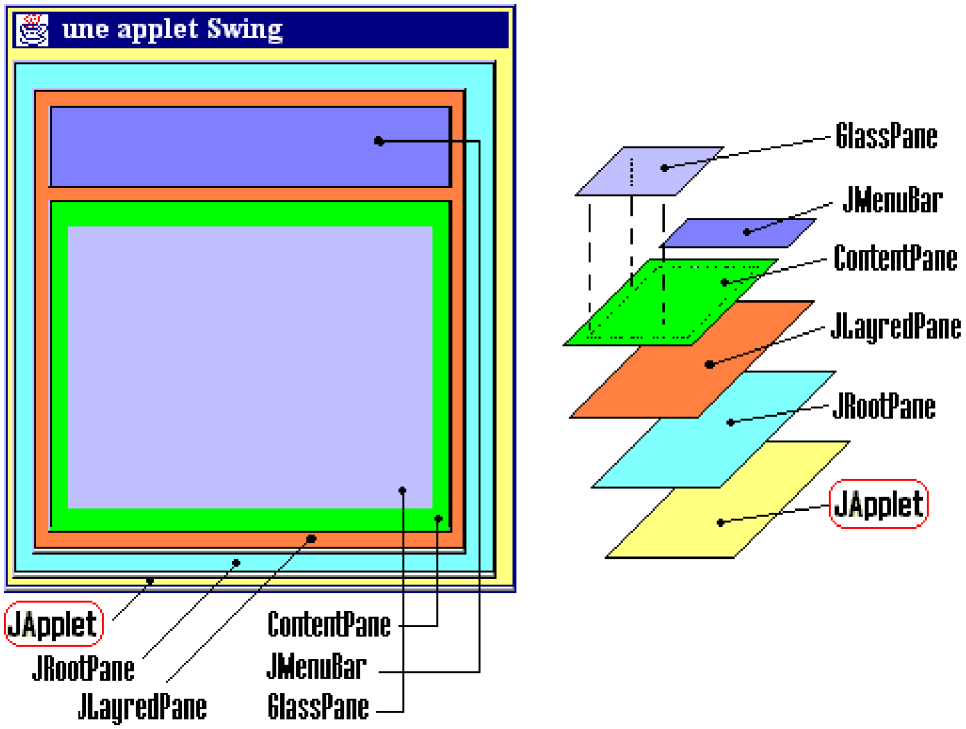
La méthode **public void paint( )** d'une applet Awt est automatiquement appelée par l'applet pour dessiner l'ensemble des graphiques de l'applet. Lorsque l'applet doit être redessinée (la première fois et à chaque fois que l'applet est masquée totalement ou partiellement) c'est la méthode **paint** qui est automatiquement appelée. Nous avons déjà appris que si nous voulions effectuer des dessins spécifiques ou des actions particulières lors du redessinement de l'applet nous devions **redéfinir la méthode paint** de l'applet.

Lorsque l'on dépose un composant sur une applet de classe **Applet**, le parent du composant est l'applet elle-même.

2. Avec la bibliothèque Swing

Le travail est plus complexe avec Swing, en effet le **JApplet** comme le **JFrame,** est un conteneur de composants visuels qui dispose de 4 niveaux de superposition d'objets à qui est déléguée la gestion du contenu du **JApplet**.

2.1 Les 4 couches superposées d'affichage d'une JApplet



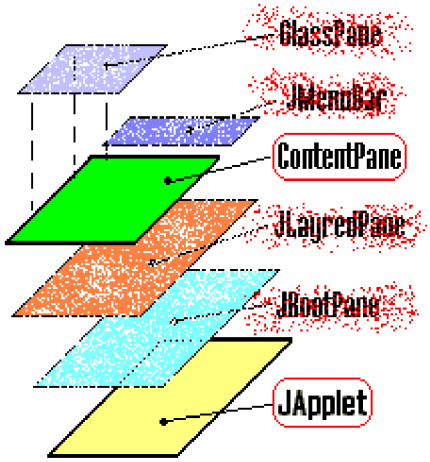
La racine JRootPane, les couches JLayredPane et GlassPane servent essentiellement à l'organisation des menus et du look and feel. Nous rappellons que seule la couche ContentPane doit être utilisée par le développeur pour écrire ses programmes.

Tout ce que l'on dépose sur une **JApplet** doit en fait l'être sur son **ContentPane** qui est une instance de la classe **Container**. C'est pourquoi dans les exemples qui sont proposés vous trouvez l'instruction :

```
this.getContentPane( ).add(....)
```

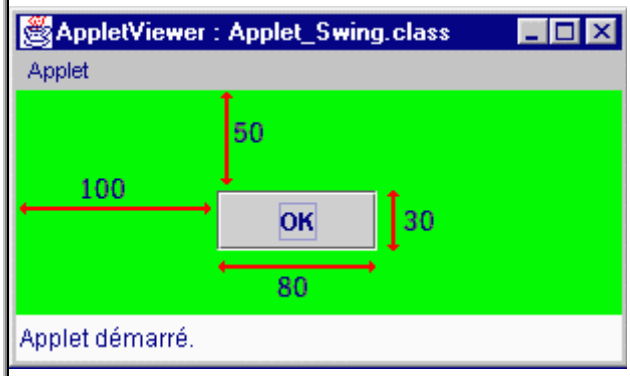
2.2 Les deux seules couches utilisées

Parmi ces couches nous n'en utiliserons que deux :



Colorons l'applet en jaune, son contentPane en vert et déposons un JButton "OK" sur l'applet (sur son contentPane) :

```
public class Applet_Swing extends JApplet
{
 Container contentPane = getContentPane();
 void Ecrire ( )
 {
  System.out.println("couleur this =" + this.getBackground( ).toString( ));
  System.out.println("couleur contentPane =" + contentPane.getBackground( ).toString( ));
  System.out.println("couleur parent du contentPane =" +
                          contentPane.getParent( ).getBackground( ).toString( ));

 }
 public void init( )
 {
   JButton bouton = new JButton("OK");
   bouton.setBounds(100,50,80,30);
   contentPane.setLayout(null);
   contentPane.add(bouton);
   contentPane.setBackground(Color.green);
   this.setBackground(Color.yellow);
   Ecrire( );
 }
}
/* les valeurs des couleurs en RGB :
yellow = 255,255,0 <--- JApplet
green = 0,255,0   <--- contentPane de JApplet

*/
```

Résultat visuel de l'applet :	Résultat des écritures de l'applet :
	couleur this = java.awt.Color[r=255,g=255,b=0] **yellow** couleur contentPane = java.awt.Color[r=0,g=255,b=0] **green** couleur parent du contentPane = java.awt.Color[r=255,g=255,b=0] **yellow**

Nous remarquons que nous voyons le bouton déposé et positionné sur le contentPane et le fond vert du contentPane, mais pas le fond de l'applet (**this**.setBackground(Color.yellow); représente la coloration du fond de l'applet elle-même).

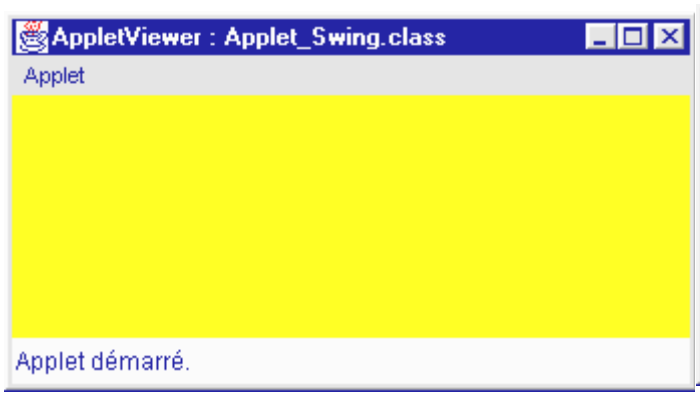
2.3 la méthode paint de l'applet redessine l'applet seulement

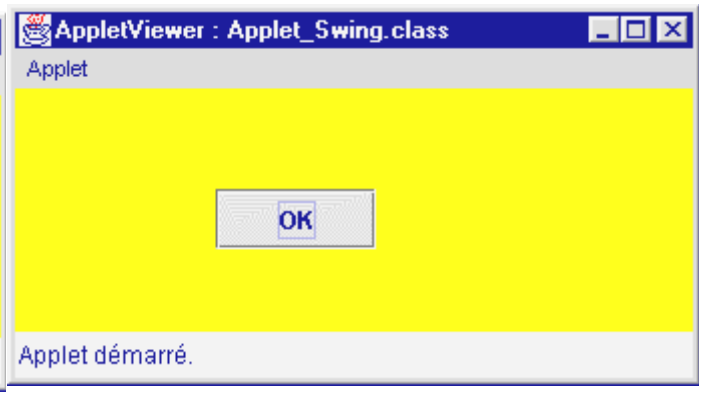
Essayons d'agir comme avec une applet Awt en redéfinissant la méthode paint (**JApplet** dérivant de **Applet** possède la méthode paint) sans mettre de code dans le corps de cette méthode, afin de

voir quel travail effectue la méthode paint :

```
public class Applet_Swing extends JApplet
{
 Container contentPane = getContentPane();
 void Ecrire ( )
 {
  System.out.println("couleur this =" + this.getBackground( ).toString( ));
  System.out.println("couleur contentPane =" + contentPane.getBackground( ).toString( ));
  System.out.println("couleur parent du contentPane =" +
                contentPane.getParent( ).getBackground( ).toString( ));

 }
 public void init( )
 {
   JButton bouton = new JButton("OK");
   bouton.setBounds(100,50,80,30);
   contentPane.setLayout(null);
   contentPane.add(bouton);
   contentPane.setBackground(Color.green);
   this.setBackground(Color.yellow);
   Ecrire( );
 }

 public void  paint (Graphics g)
 { // redéfinition de la méthode : pas d'action nouvelle
 }

}
/* les valeurs des couleurs en RGB :
yellow = 255,255,0 <--- JApplet
green = 0,255,0   <--- contentPane de JApplet

*/
```

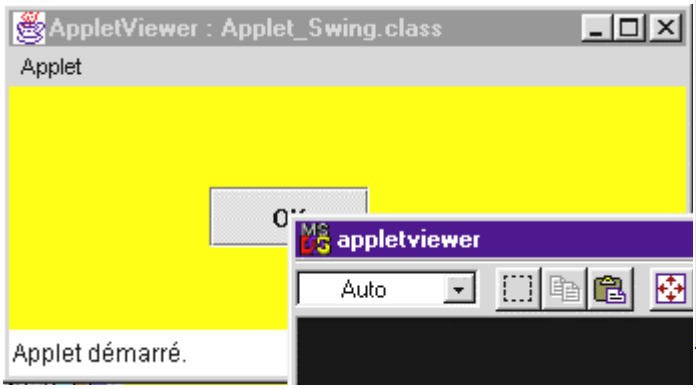
Résultats obtenus lors de l'exécution de l'applet :



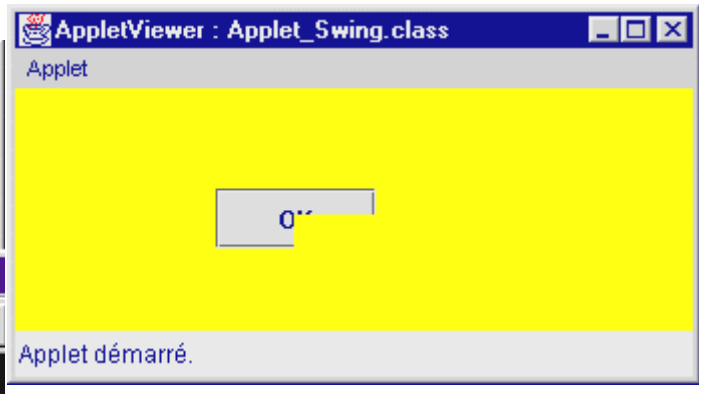


Nous nous rendons compte que cette fois-ci, le fond du contentPane vert n'a pas été affiché, mais que c'est le fond jaune de l'applet elle-même qui l'est. Le bouton OK n'est pas visible bien qu'il existe sur l'applet.

Nous masquons entièrement l'applet par une fiche, puis nous le démasquons : le bouton OK est alors redessiné par l'applet mais pas par la méthode **paint**. Si plutot nous faisons passer la souris dans la zone supposée du bouton OK c'est le bouton qui se redessine lui-même mais pas la méthode **paint**



En effet après avoir fait apparaître le bouton masquons partiellement l'applet avec une autre fenêtre, puis démasquons l'applet en rendant la fenêtre de l'applet active.



Nous voyons que l'applet a bien été rafraîchie mais que l'image du bouton ne l'a pas été, donc la méthode **paint** redéfinie redessine l'applet, mais n'a pas redessiné le bouton OK qui est déposé sur le contentPane.

Conclusion n°1
La méthode **paint** redéfinie redessine l'applet, mais ne redessine pas les objets du contentPane.

2.4 Variations : appel à la méthode paint de la super classe

1°) Sachant qu'une applet de classe **Applet** de Awt se redessine avec tous ses composants :

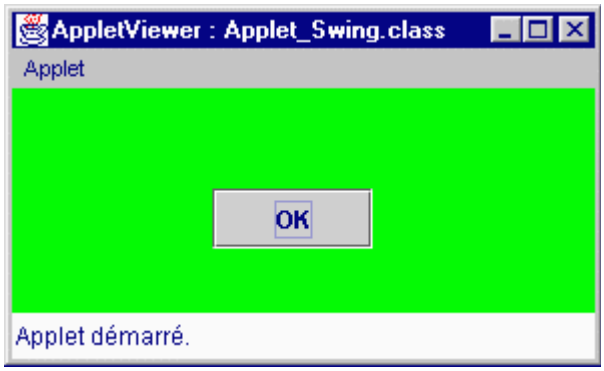
java.awt.Container
|
+--java.awt.Panel
|
+--java.applet.Applet

2°) Sachant que **JApplet** de Swing est une classe fille de **Applet** , nous nous proposons alors de faire appel à la méthode **paint** de l'ancêtre (la super classe) qui est la classe **Applet** de Awt. Dans ce cas nous forcerons l'applet Swing à réagir comme une applet Awt :

java.applet.Applet
|
+--javax.swing.JApplet

```
public void  paint (Graphics g)
{ // redéfinition de la méthode : on appelle la méthode de l'ancêtre
    super.paint ( g );
}
```

Nouveau résultat d'exécution :



Nous remarquons que l'appel à la méthode **paint** de la super classe provoque un affichage correct, le masquage et le démasquage fonctionnent correctement aussi. C'est cette solution que nous avons adoptée dans les exercices corrigés.

Problème potentiel d'instabilité de Java

> Les concepteurs de Swing déconseillent cette démarche car la surcharge de la méthode **paint** de l'applet (en fait de la classe **Container**) pourrait interférer dans certains cas avec la méthode **paint** du **JComponent** qui surcharge elle-même déjà la méthode **paint** de la classe **Container**.

3. Méthode paintComponent pour utiliser le redessinement

3.1 Généralités pour un composant

Il est conseillé en général d'utiliser systématiquement la méthode **paintComponent** de chaque composant qui est appelée automatiquement dès que l'applet est redessinée. En fait chaque composant reçoit une notification de redessinement et la méthode **paintComponent** de chaque composant est exécutée. Si en outre nous voulons effectuer des actions spécifiques pendant le redessinement d'un composant, nous devons alors surcharger la méthode **paintComponent** de ce composant.

3.2 Mettez un JPanel dans votre applet

Le composant de base qui encapsule tous les autres composants d'une applet **JApplet** est son **contentPane**. Etant donné que contentPane est un objet nous ne pouvons donc pas surcharger la méthode du **contentPane** (on peut surcharger une méthode dans une classe) nous créons une nouvelle classe héritant des **JPanel** :

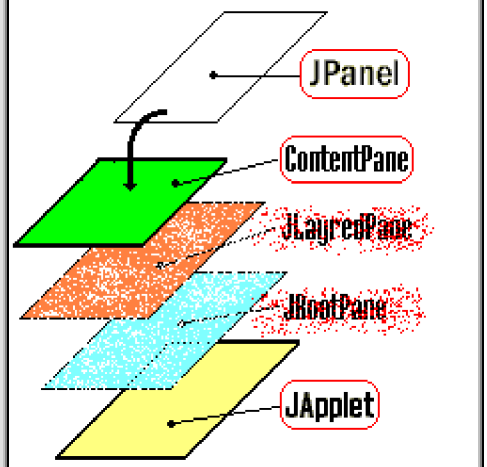
```
class JPanelApplet extends JPanel  {    //-- constructeur :
    JPanelApplet( )      {
      setBackground(Color.white); // le fond est blanc
    }
}
```

Nous allons rajouter une couche supplémentaire sur l'applet avec un nouvel objet instancié à partir de cette nouvelle classe héritant des **JPanel** dont nous surchargerons la méthode **paintComponent** (puisqu'elle est invoquée automatiquement par l'applet). Nous déposerons ce **JPanel** sur le **contentPane** de l'applet.

```
class JPanelApplet extends JPanel  {
   JPanelApplet( ) // constructeur
  {
        setBackground(Color.white); // le fond est blanc
   }
  public void paintComponent (Graphics g)
  { // le redessinement est traité ici
    super.paintComponent(g);
    g.drawString("Information dessinée sur le fond graphique", 10, 40);
   }
}
```

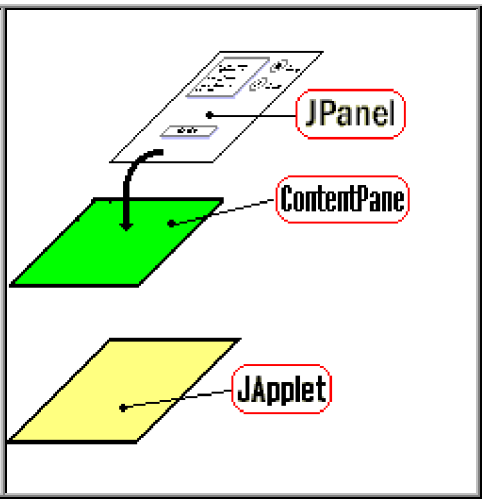
Nous instancions un objet panneau de cette classe et l'ajoutons au **contentPane** de l'applet :

```
public class Applet_AfficheSwing extends JApplet {
  JPanelApplet panneau;

  public void init()
 {
   Container contentPane = getContentPane( );
   panneau = new JPanelApplet( );
   contentPane.add(panneau);
   contentPane.setBackground(Color.green);
   this.setBackground(Color.yellow);
  }
}
```

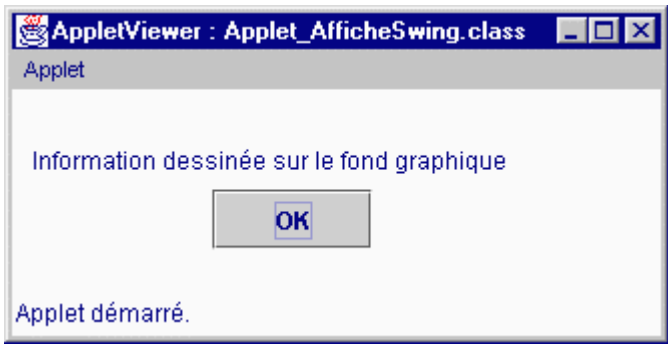


Nous déposerons tous nos composants sur ce **JPanel**, ceci permettra aussi de transformer plus facilement des applications encapsulées dans un **JPanel** qui peut être identiquement déposé sur un **JFrame** vers une applet Swing..

```
public class Applet_AfficheSwing extends JApplet {
  JPanelApplet panneau;

  public void init( )  {
   Container contentPane = getContentPane( );
   panneau = new JPanelApplet( );
   contentPane.add(panneau);
   contentPane.setBackground(Color.green);
   this.setBackground(Color.yellow);
   JButton bouton = new JButton("OK");
   bouton.setBounds(100,50,80,30);
   panneau.setLayout(null);
   panneau.add(bouton);
  }
}
```



Voici le résultat de l'exécution de l'applet ainsi construite:



Conclusion

Toute action à effectuer lors du redessinement de l'applet pourra donc être intégrée dans la méthode surchargée **paintComponent** de la classe **JPanelApplet**, le code minimal d'une applet Swing ressemblera alors à ceci :

```
class JPanelApplet extends JPanel
 {
   JPanelApplet( ) // constructeur
  {
        .........
   }

 public void paintComponent (Graphics g)
  { // le redessinement est traité ici
     super.paintComponent(g);
     //..... vos actions spécifiques lors du redessinement
   }
}

public class Applet_AfficheSwing extends JApplet
{
  JPanelApplet panneau;

  public void init( )
 {
   Container contentPane = getContentPane( );
   panneau = new JPanelApplet( );
   contentPane.add(panneau);
   ..... // les composants sont ajoutés à l'objet "panneau"
  }
}
```