

Créer des interfaces graphiques en Java

Le paquetage java.awt

Ce paquetage fournit un ensemble de classes permettant de construire et de manipuler les interfaces graphiques.

Exemple

```
import java.awt.*;

public class Appli2 extends Frame {
    static final String message = "Hello World";
    private Font font;

    public Appli2 () {
        font = new Font("Helvetica", Font.BOLD, 48);
        setSize(400, 200);
        setVisible(true);
    }

    public static void main(String args[]) {
        Frame fr = new Appli2 ();
    }
}
```

```
public void paint(Graphics g) {  
    // Un ovale plein rose  
    g.setColor(Color.pink);  
    g.fillOval(10, 10, 330, 100);  
  
    // Un contour ovale rouge épais. Les contours  
    // java sont seulement d'épaisseur un pixel : // Il  
    // faut donc en dessiner plusieurs.  
    g.setColor(Color.red);  
    g.drawOval(10,10, 330, 100);  
    g.drawOval(9, 9, 332, 102);  
    g.drawOval(8, 8, 334, 104);  
    g.drawOval(7, 7, 336, 106);  
  
    // le texte  
    g.setColor(Color.black);  
    g.setFont(font);  
    g.drawString(message, 40, 75);  
}  
}
```

donne l'affichage



La classe Graphics

Lors de la réception d'un événement expose, un objet `Graphics` est créé par le "moteur Java". Cet objet contient et décrit tout ce qu'il faut avoir pour pouvoir dessiner ("boîtes de crayons de couleurs", les divers "pots de peinture", les valises de polices de caractères, les règles, compas pour dessiner des droites et cercles, ...) ainsi que la "toile" de dessin sur laquelle on va dessiner. Cette toile correspond à la partie qui était masquée et qui doit être redessinée.

On peut parfois récupérer le `Graphics` associé à un `Component` par la méthode `getGraphics()` de la classe `Component`.

Cette classe donne les primitives de dessin à l'aide de méthodes (qui doivent être lancées sur un objet de cette classe).

en général cet objet est l'argument de la méthode `paint()`. Cet argument a été construit par la Java virtuelle machine.

exemple :

```
public void paint(Graphics g) {
    g.drawArc(20, 20, 60, 60, 90, 180);
    g.fillArc(120, 20, 60, 60, 90, 180);
}
```

La classe Graphics (suite)

On trouve principalement :

```
clearRect(int, int, int, int)
```

efface le rectangle indiqué à l'aide de la couleur de fond courante.

```
drawArc(int x, int y, int width,  
        int height, int startAngle,  
        int arcAngle)
```

dessine un arc à l'intérieur du rectangle spécifié commençant à `startAngle` et faisant un angle de `arcAngle` avec lui.

```
drawImage(Image, int, int,  
ImageObserver)
```

Dessine l'image au coordonnées indiquées. À l'appel, le dernier argument est `this`.

```
drawLine(int x1, int y1, int x2, int  
y2)
```

Dessine une ligne entre les points (x_1, y_1) et (x_2, y_2) .

```
drawOval(int x, int y, int width, int  
height)
```

Dessine le contour de l'ellipse contenue dans le rectangle indiqué.

```
drawPolygon(int xpoints[], int  
ypoints[], int npoints)
```

dessine un polygone défini par les tableaux de points `xpoints` et `ypoints`.

La classe Graphics (suite)

`drawRect(int x, int y, int width, int height)`

dessine le contour du rectangle indiqué.

`drawString(String, int, int)`

dessine la chaîne au coordonnées indiquées (en utilisant la couleur et la police de l'instance).

`fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)`

Remplit la portion d'ellipse contenue dans le rectangle indiqué avec la couleur courante.

`fillOval(int x, int y, int width, int height)`

Remplit l'ellipse contenue dans le rectangle indiqué avec la couleur courante.

`fillRect(int x, int y, int width, int height)`

Remplit le rectangle indiqué avec la couleur courante.

`setColor(Color)`

positionne la couleur courante.

`setFont(Font)`

positionne la police courante.

Exemples : les polices

```
import java.awt.*;

public class plusPolicesApp extends Frame {
public plusPolicesApp () {
    setSize(400, 200);
    setVisible(true);
}

    public static void main(String args[]) {
Frame fr = new plusPolicesApp ();
    }

    public void paint(Graphics g) {
        Font f = new Font("TimesRoman",
Font.PLAIN, 18);
        Font fb = new Font("TimesRoman",
Font.BOLD, 18);
        Font fi = new Font("TimesRoman",
Font.ITALIC, 18);
        Font fbi = new Font("TimesRoman",
Font.BOLD + Font.ITALIC, 18);
        g.setFont(f);
        g.drawString("Times plain", 10, 25);
        g.setFont(fb);
        g.drawString("Times gras", 10, 50);
        g.setFont(fi);
        g.drawString("Times italique", 10, 75);
        g.setFont(fbi);
        g.drawString("Times gras italique", 10, 100);
    }
}
```

ce qui donne :

Times plain

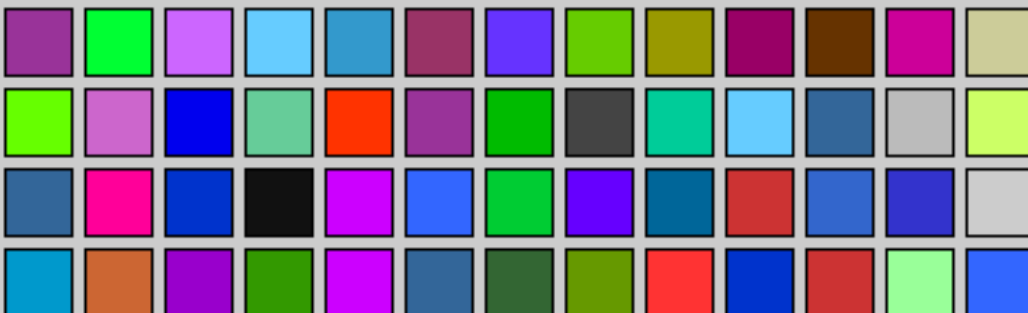
Times gras

Times italique

Times gras italique

Exemples (suite) : la couleur

```
import java.awt.*;
public class ColorBoxesApp extends Frame {
public ColorBoxesApp () {
    setSize(400, 200);
    setVisible(true);
}
    public static void main(String args[]) {
Frame fr = new ColorBoxesApp ();}
    public void paint(Graphics g) {
        int rval, gval, bval;
        for (int j = 30; j < (this.getSize().height -25); j +=
30)
            for (int i = 5; i < (this.getSize().width -25); i+=
30) {
rval = (int)Math.floor(Math.random() * 256);
gval = (int)Math.floor(Math.random() * 256);
bval = (int)Math.floor(Math.random() * 256);
g.setColor(new Color(rval,gval,bval));
g.fillRect(i,j,25,25);
g.setColor(Color.black);
g.drawRect(i-1,j-1,25,25);
            }
        }
    }
}
```



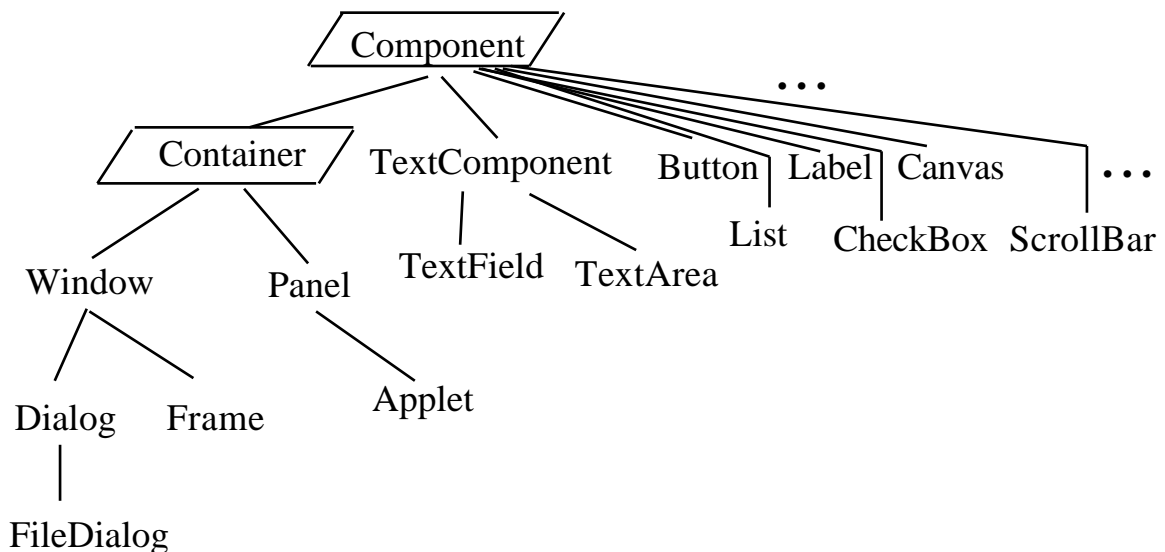
On peut utiliser les constantes de la classe `Color` comme `Color.green`, `Color.pink`, ...

Les composants graphiques

Les classes qui les traitent appartiennent au paquetage `java.awt` (Abstract Windowing Toolkit). Il fournit les objets graphiques habituels dans les interfaces :

- des objets "contrôles" (boutons, champ de texte, case à cocher, ...)
- des objets conteneurs qui gèrent leurs contenus (positionnement, ...).
- la gestion des événements

Les classes composant ce système sont :



`Component` et `Container` sont des classes abstraites. Les composants ont une réaction similaire quel que soit la plate-forme ("feel" standard). Ils ont l'aspect des objets graphiques de la plate-forme ("look" adapté).

Composants graphiques de base

classe Button

Exemple :

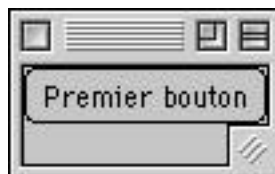
```
import java.awt.*;

public class ButtonTestApp extends Frame {

    public ButtonTestApp () {
        add("Center", new Button("Premier
bouton"));
        pack();
        setVisible(true);
    }

    public static void main(String args[ ]) {
        Frame fr = new ButtonTestApp ();
    }
}
```

L'aspect du bouton sur un macintosh est :



classe Button

constructeurs

Button(), Button(String intitule).

principales méthodes :

getLabel(), setLabel(String).

Composants graphiques de base (suite)

classe Label

constructeurs

Label(), Label(String intitule),
Label(String intitule, int alignement).

Par défaut l'alignement est à gauche. Les
valeurs possibles sont :

Label.RIGHT, Label.LEFT, Label.CENTER.

principales méthodes :

getText(), setText(String),
setAlignement(int), getAlignement().

les cases à cocher

i.e. plusieurs cases peuvent être
sélectionnées.

Ce sont des objets de la classe Checkbox.

constructeurs

Checkbox(), Checkbox(String intitule,
Checkbox(String intitule,
CheckboxGroup groupe, boolean etat).

Ce dernier constructeur permet de
positionner l'état sélectionné ou non de la
case à cocher et aussi de la faire appartenir
à un groupe de case à cocher (auquel cas elle
deviendra une case à option i.e. un bouton
radio).

Composants graphiques de base (suite)

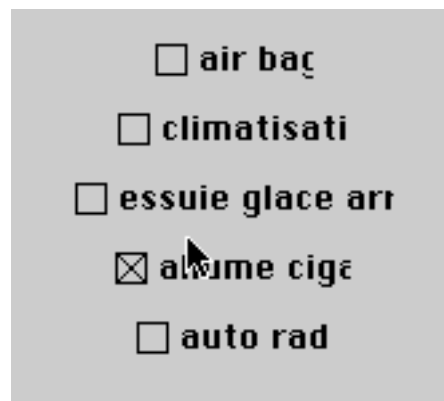
les cases à cocher (suite)

principales méthodes :

`getLabel()`, `setLabel(String)`,
`getState()`, `setState(boolean)`

```
add(new Checkbox("air bag"));  
add(new Checkbox("climatisation"));  
add(new Checkbox("essuie glace arrière"));  
add(new Checkbox("allume cigare", null, true));  
add(new Checkbox("auto radio"));
```

pour obtenir :



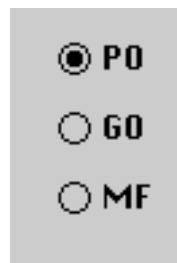
Composants graphiques de base (suite)

cases à option

On construit les mêmes composants que ci-dessus et on les met dans une boîte qui les gère de sorte à n'en sélectionner qu'un seul à la fois.

```
CheckboxGroup cbg = new CheckboxGroup();  
  
add(new Checkbox("PO", cbg, true));  
add(new Checkbox("GO", cbg, false));  
add(new Checkbox("MF", cbg, false));  
}
```

donne :



Composants graphiques de base (suite)

Bouton menu

C'est un bouton qui propose des choix lorsqu'il est actionné

Ce sont des objets de la classe `Choice`.

constructeurs

`Choice()`

principales méthodes :

`addItem()`, `getItem(int position)`

(retourne la chaîne correspondant à l'élément de la position indiquée),

`countItems()`.

exemple :

```
Choice c = new Choice();
```

```
c.addItem("Pommes");
```

```
c.addItem("Poires");
```

```
c.addItem("Scoubidou");
```

```
c.addItem("Bidou");
```

```
add(c);
```

donne :



Composants graphiques de base (suite)

champ de texte

Ce sont des objets de la classe `TextField`. C'est une zone à une seule ligne de texte à saisir. Les fenêtres de texte à plusieurs lignes sont des `TextArea`.

constructeurs

`TextField()`, `TextField(int lg_champ)`,
`TextField(String chaine)`,
`TextField(String chaine, int lg_champ)`.

principales méthodes :

`setEchoCharacter(char c)` : positionne le caractère d'écho : utile pour saisir des mots de passe.

`setEditable(boolean)` : `true` (valeur par défaut) autorise l'édition du texte par l'utilisateur du programme.

`getText()`, `setText(String)`,
`getColumns()`.

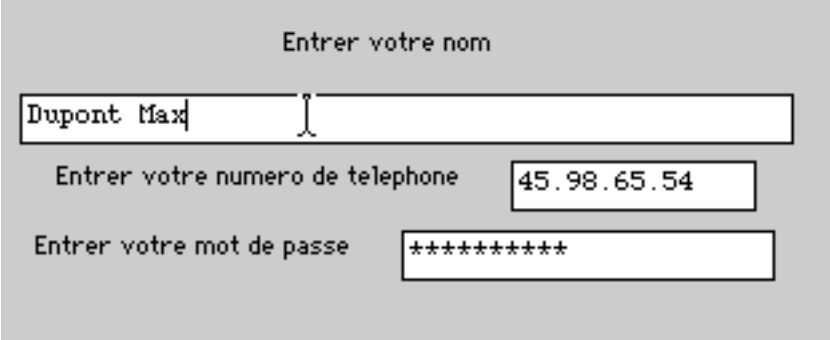
Composants graphiques de base (fin)

champ de texte (suite)

exemple :

```
add(new Label("Entrer votre nom"));
add(new TextField("votre nom ici",45));
add(new Label("Entrer votre numero de telephone"));
add(new TextField(12));
add(new Label("Entrer votre mot de passe"));
TextField t = new TextField(20);
t.setEchoCharacter('*');
add(t);
```

donne :



The screenshot shows a window titled "Entrer votre nom" with a light gray background. It contains three text input fields:

- The first field is labeled "Entrer votre nom" and contains the text "Dupont Max".
- The second field is labeled "Entrer votre numero de telephone" and contains the text "45.98.65.54".
- The third field is labeled "Entrer votre mot de passe" and contains ten asterisks "*****".

Les conteneurs

La disposition des objets dans une interface dépend de :

- l'ordre dans lequel ils ont été ajoutés dans le conteneur
- la politique de placement de ce conteneur.

AWT propose les politiques :

FlowLayout (politique par défaut dans les Panel), GridLayout, BorderLayout (politique par défaut dans les Window).

Ces politiques de positionnement sont gérées par des objets de classe `<truc>Layout`. On peut positionner une politique en écrivant :

```
||this.setLayout(new <truc>Layout());||
```

où `this` est le conteneur à gérer,
`public void setLayout(LayoutManager)`
étant une méthode de la classe `Container`.

Les politiques de placement

placement `FlowLayout`

Les composants sont ajoutés les uns à la suite des autres, ligne par ligne. Si un composant ne peut être mis sur une ligne, il est mis dans la ligne suivante.

Par défaut les composants sont centrés.



Il peuvent être alignés à gauche ou à droite :

```
||setLayout(new FlowLayout(FlowLayout.LEFT));||
```

donne



On peut indiquer les espacement haut-bas et gauche-droite pour les composants avec :

```
||setLayout(new FlowLayout(FlowLayout.LEFT, 40, 30));||
```

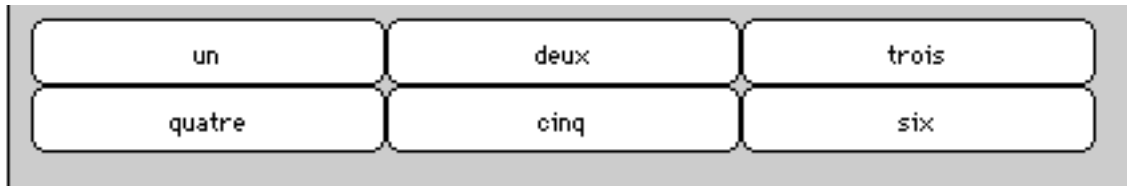
Les politiques de placement

placement GridLayout

Les composants sont rangés en lignes-colonnes et ils ont même largeur et même hauteur. Ils sont placés de gauche à droite puis de haut en bas.

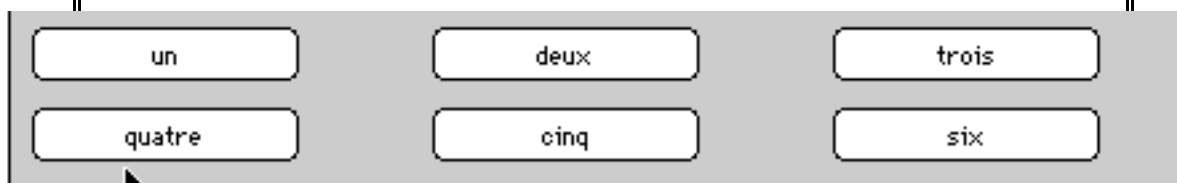
```
setLayout(new GridLayout(2,3));  
add(new Button("un"));  
add(new Button("deux"));  
add(new Button("trois"));  
add(new Button("quatre"));  
add(new Button("cinq"));  
add(new Button("six"));
```

donne



on peut aussi indiquer les espacements par :

```
setLayout(new GridLayout(2,3, 50, 10));
```



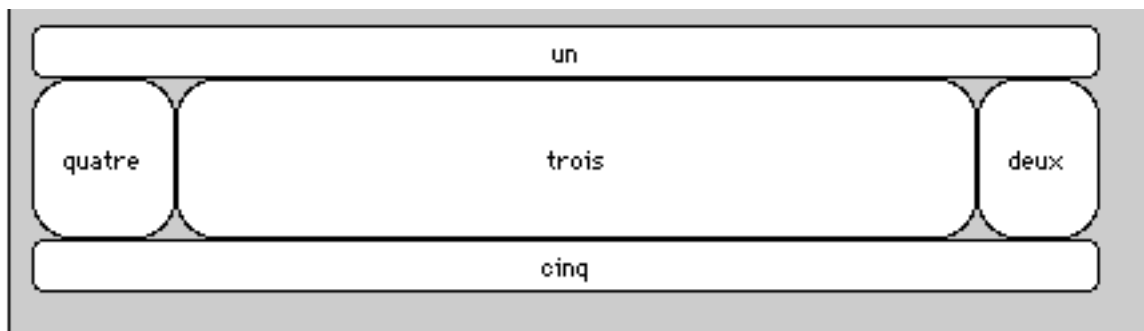
Les politiques de placement

placement BorderLayout

permet de placer les composants "sur les bords" et au centre. On indique la politique de positionnement, puis chaque fois qu'on ajoute un composant on indique où le placer :

```
setLayout(new BorderLayout());  
add("North", new Button("un"));  
add("East", new Button("deux"));  
add("Center", new Button("trois"));  
add("West", new Button("quatre"));  
add("South", new Button("cinq"));
```

donne :



On peut aussi utiliser les constantes de la classe BorderLayout et écrire :

```
add(new Button("un"), BorderLayout.NORTH);  
add(new Button("deux"), BorderLayout.EAST);  
add(new Button("trois"), BorderLayout.CENTER);  
add(new Button("quatre"), BorderLayout.WEST);  
add(new Button("cinq"), BorderLayout.SOUTH);
```

La méthode `add()`

Cette méthode (de la classe `Container`) permet d'ajouter des composants graphiques dans un conteneur.

Si ce conteneur est "géré par un `BorderLayout`", cette méthode doit avoir 2 arguments : l'emplacement (de type `String`) et l'objet à ajouter.

Si ce conteneur est "géré par un `FlowLayout` ou un `GridLayout`", cette méthode doit avoir un seul argument : l'objet à ajouter.

Le positionnement en absolu

On désire parfois positionner les composants dans un container sur des coordonnées bien définies et leur donner une taille déterminée.

C'est possible en indiquant au conteneur de ne pas utiliser de `LayoutManager` par :

```
||leConteneur.setLayout(null); ||
```

Par la suite on fixe les valeurs de "géométrie" du composant par :

```
||leComposant.setBounds(x, y, largeur, hauteur); ||
```

et on ajoute ce composant dans le conteneur par :

```
||leConteneur.add(leComposant); ||
```

Cette politique va plutôt à l'encontre de la politique des `LayoutManager` mais elle est souvent adoptée par les générateurs d'interfaces graphiques !! (dont certains permettent de transformer le résultat obtenu par une gestion en `GridBagLayout`).

Créer une interface graphique

Afin de manipuler des parties d'une interface en un seul bloc, ou parfois parce qu'il n'est pas possible de mettre plus d'un composant graphique directement à un emplacement donné, on constitue (et on programme !!) des regroupements. Ainsi créer des interfaces graphiques c'est "mettre des boîtes dans des boîtes dans des boîtes ...".

En Java, la "boîte de regroupement" est (évidemment) un `Container` et c'est souvent un `Panel`.

Par exemple si on veut mettre quatre Boutons poussoir dans l'emplacement `South` d'un `Container` géré par un `BorderLayout`, on écrit :

```
Panel p = new Panel();
Button bt[] = new Button[4];
for (int i = 0; i < 4; i++) {
    bt[i] = new Button(""+i);
    p.add(bt[i]);
}
le_container.add(p, BorderLayout.SOUTH);
```

Les Menus

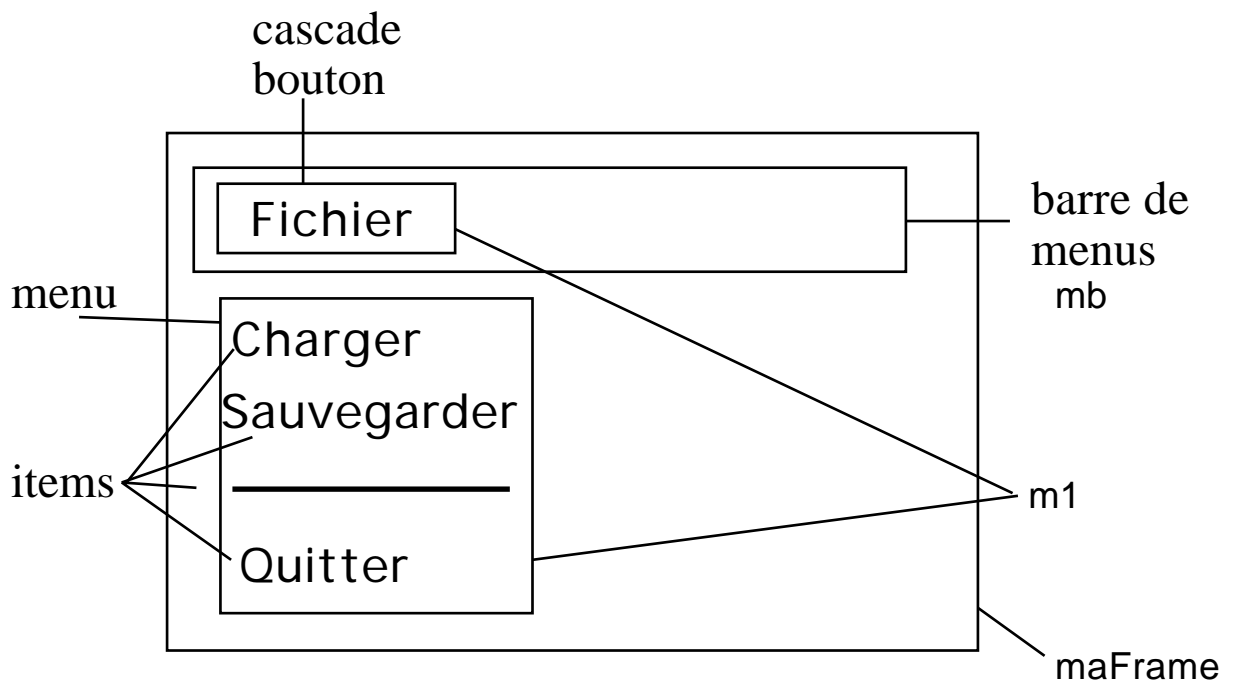
Ils ne peuvent être mis que dans des frames. On utilise la méthode `setMenuBar(MenuBar)` de la classe `Frame`. Le code à écrire est, par exemple :

```
MenuBar mb = new MenuBar();

Menu m1 = new Menu("Fichier");
m1.add(new MenuItem("Charger"));
m1.add(new MenuItem("Sauvegarder"));
m1.addSeparator();
m1.add(new MenuItem("Quitter"));

mb.add(m1);

maFrame.setMenuBar(mb);
```



Les méthodes graphiques

```
repaint(), update(Graphics g),  
paint(Graphics g)
```

Ces trois méthodes sont des méthodes de la classe `Component`. Certaines sont parfois spécifiées (= redéfinir) dans les sous classe (voir ci dessous).

`repaint()` est une méthode (qu'il ne faut jamais redéfinir) "système Java" gérée par "la thread AWT" qui appelle, dès que cela est possible, la méthode `update(Graphics g)` qui appelle ensuite `paint(Graphics g)`. `update(Graphics g)` de la classe `Component` efface le composant (le redessine avec sa couleur de fond), puis appelle `paint(Graphics g)`.

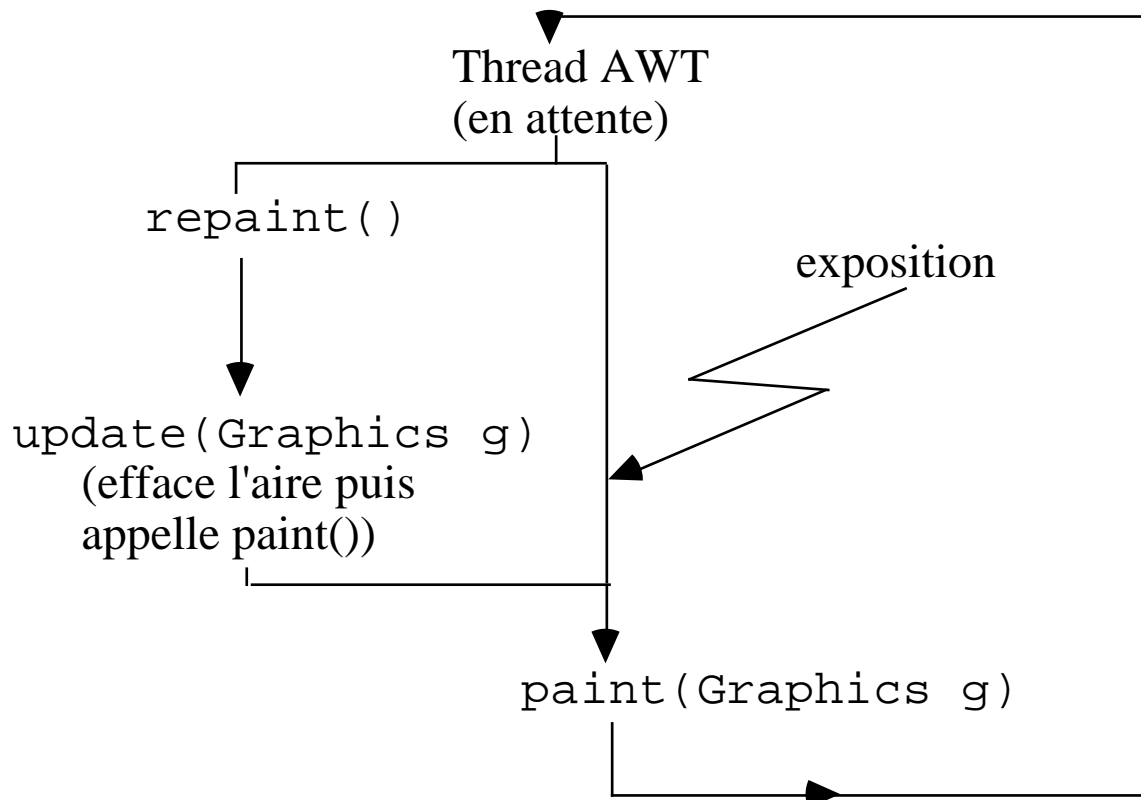
D'ailleurs `update(Graphics g)` de la classe `Component` est :

```
public void update(Graphics g) {  
    g.setColor(getBackground());  
    g.fillRect(0, 0, width, height);  
    g.setColor(getForeground());  
    paint(g);  
}
```

Le `Graphics` repéré par la référence `g` des méthodes `update()` et `paint()` a été construit par la thread AWT.

Les méthodes graphiques (suite)

`repaint()`, `update()`, `paint()`



Lors d'un événement d'exposition, `paint()` est lancé sur la partie du composant graphique qui doit être redessiné : zone de clipping.

Les animations

Très souvent la méthode `update(Graphics g)` donné par Java dans la classe `Component`, pose problème dans les animations (tremblements), car il est intercalé entre 2 dessins une image de couleur unie.

Première solution : redéfinir `update()`

on écrit dans `update()` le seul appel à `paint()`.

Dans le code ci dessus on ajoute simplement :

```
public void update(Graphics g) {  
    paint(g);  
}
```

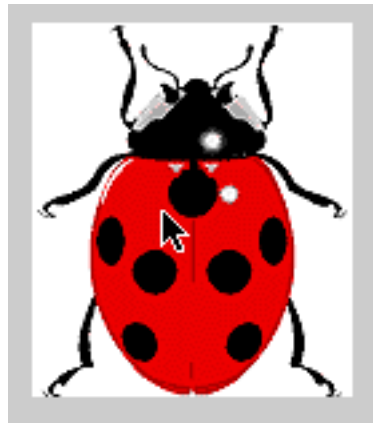
et il n'y a plus de tremblements.

Autre solution le double buffering (voir plus loin).

Les images dans les applications Java

Il y a 2 méthodes `getImage()` en Java. L'une appartient la classe `Toolkit` et c'est la seule utilisable dans une application Java autonome.

```
import java.awt.*;
public class LadyBug extends Frame {
    Image bugimg;
    public static void main(String args[ ]) {
        Toolkit tk = Toolkit.getDefaultToolkit();
        Frame fr = new LadyBug(tk, "Coccinelle");
        fr.setSize(200, 200);
        fr.setVisible(true);
    }
    public LadyBug (Toolkit tk, String st) {
        super(st);
        bugimg = tk.getImage("ladybug.gif");
    }
    public void paint(Graphics g) {
        g.drawImage(bugimg,10,10,this);
    }
}
```



De AWT à Swing

Swing est l'autre bibliothèque de composants intégrés dans le JDK à partir de Java 1.2 (et pouvant être ajouté au JDK 1.1). Au delà de fournir des composants plus riche (ce qui est déjà énorme), Swing manipule des notions très avancés et donc nécessite un cours a lui seul.

On peut néanmoins transformer facilement les programmes décrits dans ce chapitre en programmes Swing en faisant déjà les premières manipulations suivantes :

- importer le paquetage `javax.swing` (et enlever le paquetage `java.awt`)
- préfixer les classes AWT par un J (comme Java) : par un exemple un `Button` devient un `JButton`, ...
- mettre les composants et positionner le layout d'une `Frame` dans le `ContentPane` de la `Frame` et non pas directement. On écrit donc

```
|| JFrame laJFrame ...  
|| laJFrame.getContentPane().setLayout(...);  
|| laJFrame.getContentPane().add(...);  
||
```

De plus en Swing les `JFrame` se ferment automatiquement lorsqu'on clique sur leur case de fermeture.

Il est fortement déconseillé d'avoir dans une même application des composants AWT et des composants Swing.

Les événements en Java

1.1 (1.2)

C'est un des points qui a été complètement reconçu lors du passage de Java 1.0 à Java 1.1.

En 1.1, la programmation est nettement améliorée. Voir en appendice la programmation des événements en Java 1.0.

En Java 1.1, on utilise une programmation par délégation : un objet d'une classe est associé au composant graphique et est chargé (délégué) de l'exécution du code qui doit être lancé lors de la manipulation de ce composant par l'utilisateur. Il y a ainsi une nette coupure dans le code Java entre le code interface graphique et le code applicatif.

L'objet associé au composant "est à l'écoute" de ce composant : c'est un objet d'une classe qui implémente un "listener".

Les événements

Essentiellement on écrit du code comme :

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

...

Button bt = new Button("Mon Bouton");
EcouteurBouton ecouteur = new EcouteurBouton();
bt.addActionListener(ecouteur);
...

class EcouteurBouton implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Coucou");
    }
}
```

Les classes de composants graphiques possèdent des méthodes

`addXXXListener (XXXListener)`

`XXXListener` est une interface et l'argument de `addXXXListener` peut (évidemment) référencer tout objet d'une classe qui implémente cet interface.

Ce sont dans les interfaces `XXXListener` que l'on sait quelles méthodes

implémenter. Ces méthodes sont de la forme `public void nomMethode (XXXEvent e)`

L'objet référencé par `e` est un événement et a été initialisé par "le système Java" lors de la production de l'événement. On peut alors avoir les renseignements de la situation lors de la production de cet événement. Entre autre le composant graphique qui a été manipulé par l'utilisateur est connu par `getSource()`.

Un premier exemple

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

public class ButtonTestApp extends Frame
implements ActionListener {

public ButtonTestApp () {
    Button bt = new Button("Premier bouton");
    add("Center", bt);
    bt.addActionListener(this);
    pack();
    setVisible(true);
}

    public static void main(String[ ] args) {
Frame fr = new ButtonTestApp ();
    }

public void actionPerformed(ActionEvent e) {
    System.out.println("Coucou");
}
}
```

Remarques (cf. ci dessus)

Très souvent

1°) pour les petits programmes, l'écouteur des objets d'une interface graphique est l'interface graphique elle-même.

2°) il est inutile de référencer l'objet ecouteur.

Un second exemple

```
import java.awt.*;
import java.awt.event.*;
public class DeuxBoutons extends Frame {
public DeuxBoutons () {
    Button btcoucou = new Button("Coucou");
    add("Center", btcoucou );
    btcoucou .addActionListener(new
EcouteCoucou());
    Button btfin = new Button("Fin");
    add("East", btfin );
    btfin .addActionListener(new EcouteFin());
    pack(); setVisible(true);
}
    public static void main(String[ ] args) {
Frame fr = new DeuxBoutons ();
    }
}
class EcouteCoucou implements ActionListener
{
public void actionPerformed(ActionEvent e) {
    System.out.println("Coucou");
}
}
class EcouteFin implements ActionListener {
public void actionPerformed(ActionEvent e) {
    System.exit(0);
}
}
```



Fermer une fenêtre

... lors d'un clic sur la case de fermeture.

Il faut, en Java, le coder en associant un `WindowListener` par :

```
Frame maFrame = new Frame();
maFrame.addWindowListener(new MonEcouteurFenetre
());
```

avec la classe `MonEcouteurFenetre` ci dessous :

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

public class MonEcouteurFenetre implements
WindowListener {

    /* appelé avant que la fenêtre soit fermée. On
    peut ainsi outrepasser cette fermeture, faire des
    sauvegardes, ... Ce n'est pas le cas ici.
    */
    public void windowClosing(WindowEvent e) {
        e.getWindow().dispose();
        // ou pour terminer complètement
        // l'application :
        // System.exit(0);

    }
}
```

```
/* invoqué après que la fenêtre ait été fermée */  
public void windowClosed(WindowEvent e) { }  
  
public void windowOpened(WindowEvent e) { }  
  
public void windowIconified(WindowEvent e) { }  
public void windowDeiconified(WindowEvent e){ }  
public void windowActivated(WindowEvent e) { }  
public void windowDeactivated(WindowEvent e) {  
}  
  
}
```

Classe adaptateur

Lorsqu'une interface listener demande d'implanter beaucoup de méthodes, Java propose des classes qui donnent un code par défaut à ces méthodes. Une telle classe est appelée un adaptateur. C'est le cas par exemple pour un `WindowListener` qui propose la classe adaptateur `WindowAdaptor`. Il suffit alors pour construire un listener de dériver de cette classe adaptateur.

Dans l'exemple précédent, il suffit d'écrire :

```
public class MonEcouteurFenetre extends WindowAdaptor {  
    /* appelé avant que la fenêtre soit fermée. On peut ainsi  
    outrepasser cette fermeture, faire des sauvegardes, ... Ce  
    n'est pas le cas ici.  
    */  
    public void windowClosing(WindowEvent e) {  
        e.getWindow().dispose();  
        // ou pour terminer complètement  
        //l'application :  
        // System.exit(0);  
    }  
}
```

Remarque :
cela est très pratique mais ne fonctionne pas si on veut que notre listener hérite aussi d'une autre classe (pas d'héritage multiple).

Classe anonyme, classe interne

Java 1.1 (et suivant) propose des avancés syntaxiques pour définir des classes. Ce sont :

- les classes anonymes
- les classes internes

Il se trouve que ces particularités du langage sont beaucoup utilisées pour les traitement des événements.

Les classes anonymes

On peut définir une classe « à la volée », sans lui donner de nom en précisant simplement la classe dont elle dérive ou l'interface qu'elle implémente.

En fait une classe anonyme est créée syntaxiquement au même endroit que son instance. La syntaxe est donc :

```
new classeOuInterfaceParent (listArguments) {  
    corpsDeLaClasseAnonyme  
}
```

Très souvent cet objet construit "à la volée" est lui même argument d'une méthode.

Le second exemple avec classes anonymes

```
import java.awt.*;
import java.awt.event.*;
public class DeuxBoutons extends Frame {
    public DeuxBoutons () {
        Button btcoucou = new
Button("Coucou");
        add("Center", btcoucou );
        btcoucou .addActionListener(new
ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("Coucou");
            }
        });

        Button btfm = new Button("Fin");
        add("East", btfm );
        btfm .addActionListener((new
ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.exit(0);
            }
        } ));
        pack(); setVisible(true);
    }

    public static void main(String[] args) {
        Frame fr = new DeuxBoutons ();
    }
}
```

Classe anonyme : conclusion

Les classes anonymes sont particulièrement adaptées lorsque :

- cette classe possède un corps très court
- une seule instance de la classe est nécessaire
- la classe est utilisée juste après sa définition
- le nom de la classe apporte très peu à la compréhension du code.

Classe interne

C'est une classe définie à l'intérieur d'une autre classe. Son énorme avantage est de pouvoir utiliser facilement les champs de la classe englobante.

Si on n'utilise pas les classes internes et que l'on veut, dans une classe externe, utiliser les champs d'une autre classe, il faut référencer ces champs, passer ces références à la classe externe comme argument de constructeur(s) de la classe externe et se servir de ces arguments pour initialiser des champs de la classe externe :

```
class A {
    int champAUtiliserDansAutreClasse;
    ...
    ... new B(champAUtiliserDansAutreClasse) ...
}

class B {
    int repereA;
    B(int pouRecupereraDeA) {
        repereA = pouRecupereraDeA;
    }
    void methUtiliseChampdeA() { repereA = ... }
    ...
}
```


Classe interne (suite)

Avec les classes internes, la syntaxe est plus souple :

```
class A {  
    int champAUtiliserDansAutreClasse;  
    ...  
  
    class BInterneaA {  
        void methUtiliseChampdeA() {  
            champAUtiliserDansAutreClasse = ...  
        }  
        ...  
    }  
}
```

Remarque :

- une classe anonyme est un exemple de classe interne.

Le second exemple avec classes internes

```
import java.awt.*;
import java.awt.event.*;
public class DeuxBoutons extends Frame {
    Button btcoucou, btfin;
    public DeuxBoutons () {
        btcoucou = new Button("Coucou");
        add("Center", btcoucou );
        btcoucou .addActionListener(new
    traiteBoutons ());
        btfin = new Button("Fin");
        add("East", btfin );
        btfin .addActionListener(new traiteBoutons
    ());
        pack(); setVisible(true);
    }
    public static void main(String[ ] args) {
    Frame fr = new DeuxBoutons ();
        }

    class traiteBoutons implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        if (((Button)e.getSource()) == btcoucou) {
            System.out.println("Coucou");
        } else if (((Button)e.getSource()) == btfin) {
            System.exit(0);
        }
    }
    }
}
```

Pour se rendre compte de l'intérêt de la classe interne, réécrivez un tel code avec une seule classe `traiteBoutons` externe.

Appendice :

Le mécanisme

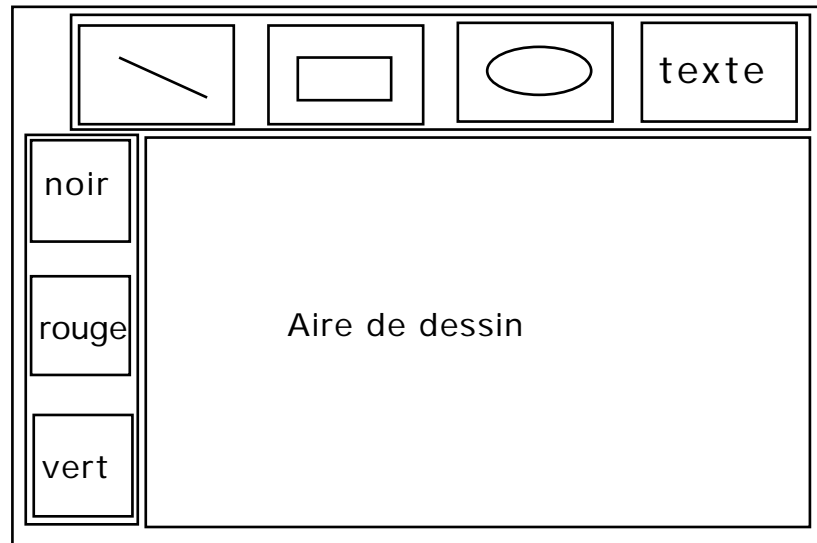
des

événements

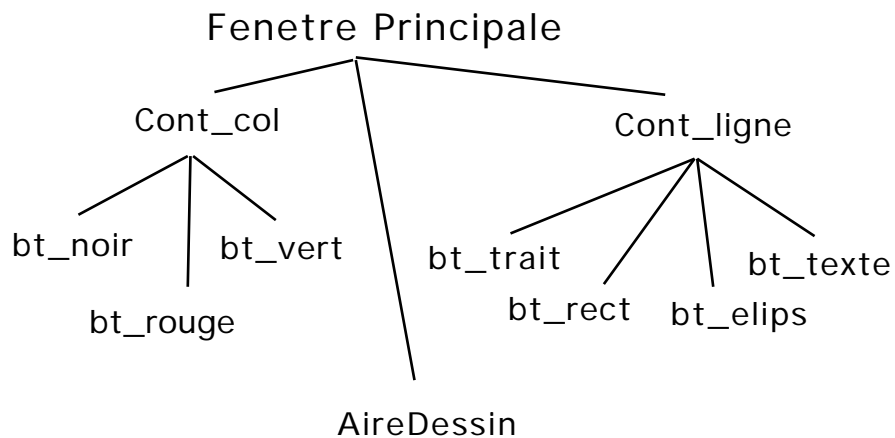
en Java 1.0

Gestion des événements en Java 1.0

L'interface graphique

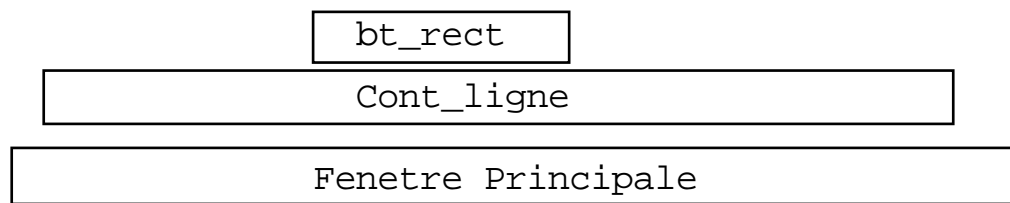


correspond à l'arborescence :

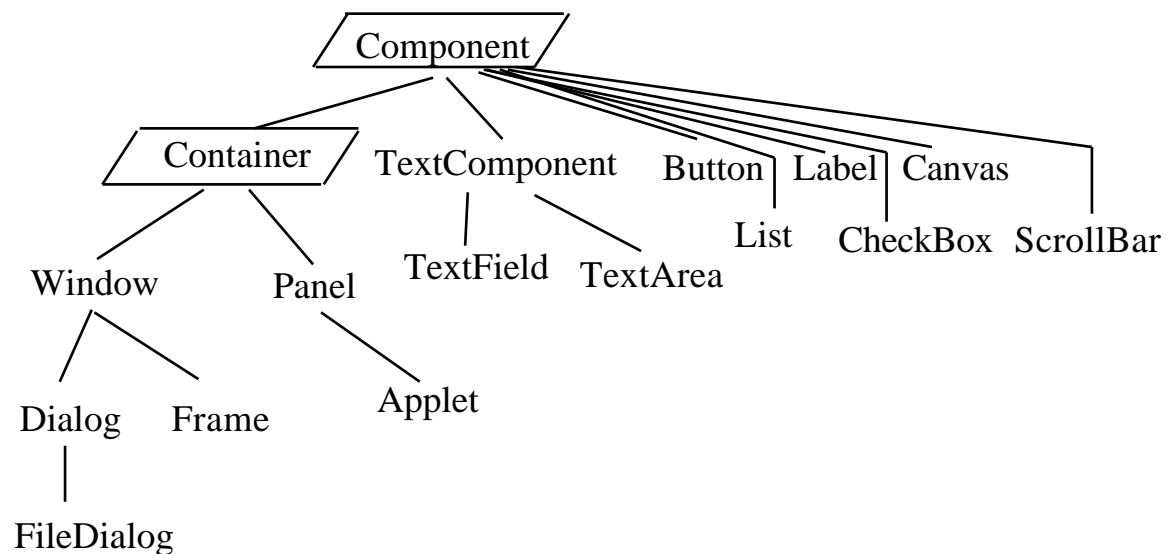


La gestion des événements (suite)

Un clic sur le bouton `bt_rect` :



L'arborescence des classes graphiques est :



La gestion des événements (suite)

Lorsque l'utilisateur agit sur un composant graphique, le système Java génère un objet (événement) de la classe `Event`. Le système Java appelle alors (i.e. envoie le message) `handleEvent(Event)` sur ce composant (i.e. ce composant lance sa méthode `handleEvent(Event)`). Si cette méthode retourne `false`, alors l'événement est envoyé au composant graphique qui le contient et le processus est relancé sur ce nouveau composant avec le même événement.

Cet objet `evt` de la classe `Event` contient entre autre le champ `id` qui indique le type de l'événement. Par exemple un clic sur un bouton génère finalement un événement dont le champ `id` vaut `Event.ACTION_EVENT`. Il possède aussi un champ `target` indiquant qui est une référence sur le composant graphique qui a reçu l'événement.

La gestion des événements (suite)

En général le composant graphique Java qui a généré (à l'aide du système Java) l'événement est un composant de base de Java : un objet de la classe `Button` ou `TextField` ou ... Or il n'y a pas de méthode `handleEvent()` dans ces classes. Par contre cette méthode est définie dans la classe `Component` mère de ces classes et c'est cette méthode qui est lancée (polymorphisme).

La méthode `handleEvent()` de la classe `Component` est :

```
public boolean handleEvent(Event evt) {
    switch (evt.id) {
        case Event.MOUSE_ENTER :
            return mouseEnter(evt, evt.x, evt.y);
        case Event.MOUSE_EXIT :
            return mouseExit(evt, evt.x, evt.y);
        case Event.MOUSE_MOVE :
            return mouseMove(evt, evt.x, evt.y);
        case Event.MOUSE_DOWN :
            return mouseDown(evt, evt.x, evt.y);
        case Event.MOUSE_DRAG :
            return mouseDrag(evt, evt.x, evt.y);
        case Event.MOUSE_UP :
            return mouseUp(evt, evt.x, evt.y);
        case Event.KEY_PRESS :
        case Event.KEY_ACTION :
            return keyDown(evt, evt.key);
        case Event.ACTION_EVENT :
            return action(evt, evt.arg);
    }
    return false;
}
```

handleEvent () de Component

De nouveau les méthodes de la classe Component appelées par `handleEvent()` (i.e. `action()`, `mouseUp()`, ...) ont simplement pour corps :

```
||return false;||
```

Et donc si ces méthodes non pas été redéfinies (par polymorphisme) dans les objets graphiques de l'interface et c'est quasiment toujours le cas puisque ces objets sont ceux des classes fondamentales de Java, l'événement est envoyé à l'objet graphique qui le contient (`return false;`) et celui-ci refait le même traitement (appel à `handleEvent()` puis aux méthodes spécifiques `action()`, `mouseUp()`, ...) avec le même objet `evt` (champ `target` inchangé) et ainsi de suite jusqu'à l'objet applet ou Fenetre Principale du programmeur c'est à dire l'objet interface graphique.

handleEvent () du programmeur

C'est donc dans cet objet, défini par le programmeur, que sont redéfinies les méthodes `handleEvent ()` ou `mouseUp ()`, ...

Cette méthode `handleEvent (Event)`

```
|| public boolean handleEvent(Event evt) { ... } ||
```

que le programmeur redéfinit, étudie les événements pouvant survenir dans l'interface. Son code est essentiellement des `if else` consécutifs. Il faut terminer cette fonction par :

```
|| return super.handleEvent(evt); ||
```

pour relancer la méthode `handleEvent ()` de la classe `Component` qui lancera d'autres méthodes plus spécifiques (`action ()`, `mouseExit ()`, `mouseDrag ()`) : en effet la méthode `handleEvent ()` de l'objet interface graphique construit par le programmeur a masqué par polymorphisme la méthode `handleEvent ()` de la classe `Component`.

La méthode `action()`

De même, le programmeur a souvent besoin de définir :

```
public boolean action(Event evt, Object arg) { ... }
```

comme méthode de son interface graphique.

Si on estime que le traitement fait par ce composant est suffisant ces fonctions doivent retourner `true` (i.e. "ça suffit"). Ainsi l'événement ne sera pas traité par le composant graphique conteneur. Sinon retourner `false`.

Cette méthode `action()` est lancée lorsque l'utilisateur a manipulé certain composant graphique. Dans ce cas, le second argument `arg` de `action()` est un objet qui dépend du composant graphique manipulé. On a

classe du composant graphique	type de arg	survient lorsque :	signification
Button	String	click sur le bouton	le label du bouton
Checkbox	Boolean	click dans la case	l'état de la case
Choice	String	sélection de l'item	le label du choix sélectionné
List	String	double click sur un item	label de l'item sélectionné
TextField	String	appui sur return	la chaîne contenue

Gestion des événements (suite)

Conclusion

Ces deux méthodes sont de la forme :

```
public boolean handleEvent(Event evt) {  
    switch (evt.id) {  
        case Event.LIST_SELECT:  
            ...  
        break;  
        case Event.LIST_DESELECT:  
            ...  
        break;  
        ...  
    }  
    return super.handleEvent(evt);  
}
```

De même, on a souvent :

```
public boolean action(Event evt, Object arg) {  
    if (evt.target instanceof TextField)  
        traiteEvtTextField(evt.target, arg);  
    else if (evt.target instanceof Choice)  
        traiteEvtChoice(evt.target, arg);  
    ...  
    ... }  
}
```

Un exemple complet

```
import java.awt.*;

public class BoutonsApp extends Frame {
    Button bt1, bt2;

    public static void main(String args [ ]) {
        Frame fr = new BoutonsApp ( );
    }

    public BoutonsApp ( ) {
        bt1 = new Button ("coucou");
        bt2 = new Button ("fin");
        add("West", bt1);
        add("Center", bt2);
        pack();
        show();
    }

    public boolean action (Event e, Object arg) {
        if (((String) arg).equals("coucou")) {
            System.out.println("coucou");
            return true;
        }
        else if (((String) arg).equals("fin")) {
            System.out.println("fin");
            System.exit(0);
            return true;
        }
        return false;
    }
}
```

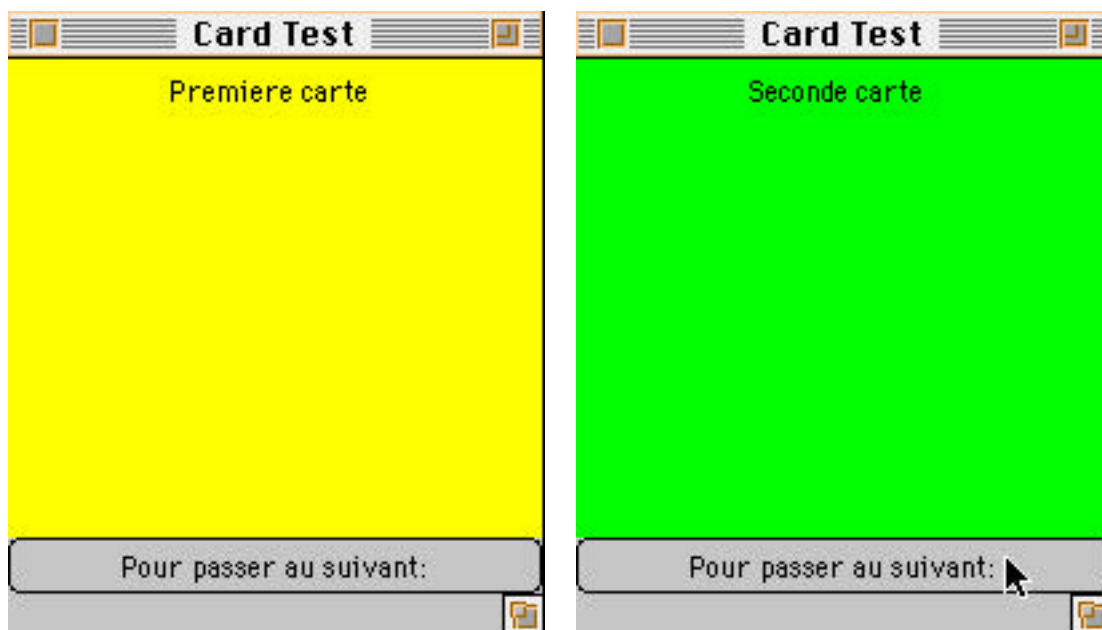
Compléments

Les politiques de placement

placement CardLayout

Un panneau menant une politique CardLayout ne fait apparaître qu'un seul de ses contenants à la fois.

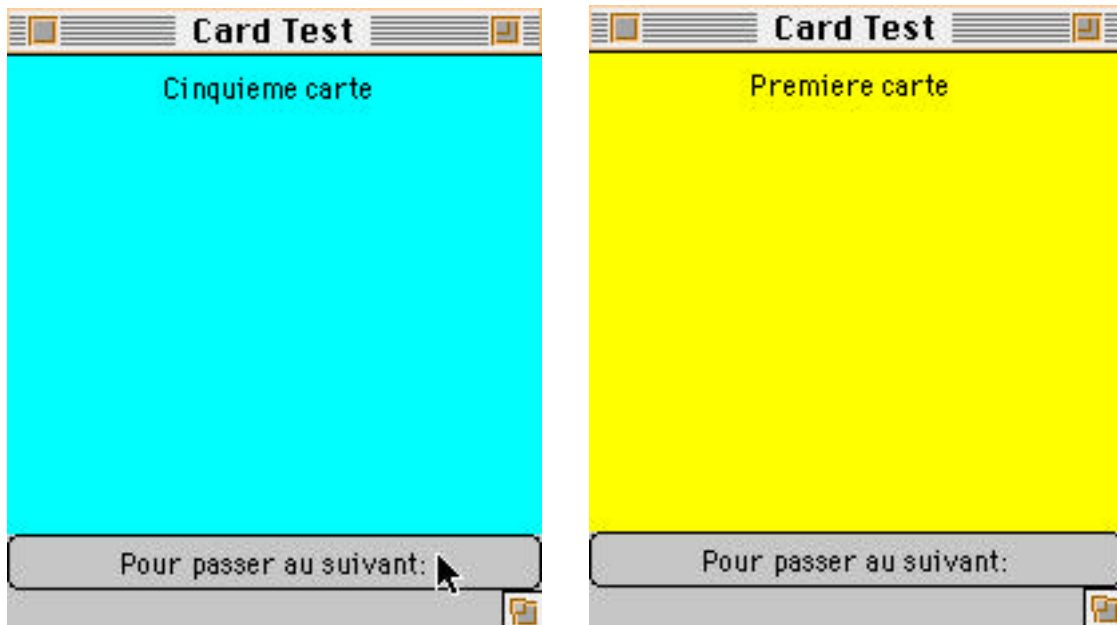
En général on crée un tel panneau panCard, on met les composants un à un dans ce panneau, puis on accède à ces éléments en lançant les méthodes `first()`, `last()`, `next()`, `previous()` sur cette instance.



Les politiques de placement (suite)

placement CardLayout (suite)

Un `next ()` sur le dernier élément amène de nouveau au premier.



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

public class CardTest implements ActionListener
{
    Panel p1, p2, p3, p4, p5;
    Label l1, l2, l3, l4, l5;
    Button monBouton;

    CardLayout myCard;
    Frame f;
    Panel panneauCentral;
```

```
public static void main (String args[]) {
    CardTest ct = new CardTest();
    ct.init();
}

public void init() {
    f = new Frame("Card Test");

    panneauCentral = new Panel();
    myCard = new CardLayout();
    panneauCentral.setLayout(myCard);

    p1 = new Panel();
    p2 = new Panel();
    p3 = new Panel();
    p4 = new Panel();
    p5 = new Panel();

    l1 = new Label("Premiere carte");
    p1.setBackground(Color.yellow);
    p1.add(l1);

    l2 = new Label("Seconde carte");
    p2.setBackground(Color.green);
    p2.add(l2);

    l3 = new Label("Troisieme carte");
    p3.setBackground(Color.magenta);
    p3.add(l3);

    l4 = new Label("Quatrieme carte");
    p4.setBackground(Color.white);
    p4.add(l4);

    l5 = new Label("Cinquieme carte");
    p5.setBackground(Color.cyan);
    p5.add(l5);
}
```

```
panneauCentral.add(p1, "First");
panneauCentral.add(p2, "Second");
panneauCentral.add(p3, "Third");
panneauCentral.add(p4, "Fourth");
panneauCentral.add(p5, "Fifth");

myCard.show(panneauCentral, "First");

f.add("Center", panneauCentral);

monBouton = new Button("Pour passer
au suivant:");
monBouton.addActionListener(this);
f.add("South", monBouton);

f.setSize(200, 200);
f.setVisible(true);
}

public void actionPerformed(ActionEvent e) {
    myCard.next (panneauCentral);
}
}
```