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 crée par le "moteur Java". Cet objet contient et décrit tout ce qu'il faut avoir pour pouvoir dessiner ("boites 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 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 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 (x1,y1) et (x2,y2).

`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)
```
Rempli la portion d'ellipse contenue dans le rectangle indiqué avec la couleur courante.

`fillOval(int x, int y,int width, int height)`
Rempli l'ellipse contenue dans le rectangle indiqué avec la couleur courante.

`fillRect(int x, int y,int width, int height)`
Rempli le rectangle indiqué avec la couleur courante.

`setColor(Color)`
positionne la couleur courante.

`setFont(Font)`
positionne la police courante.

Exemples : les polices

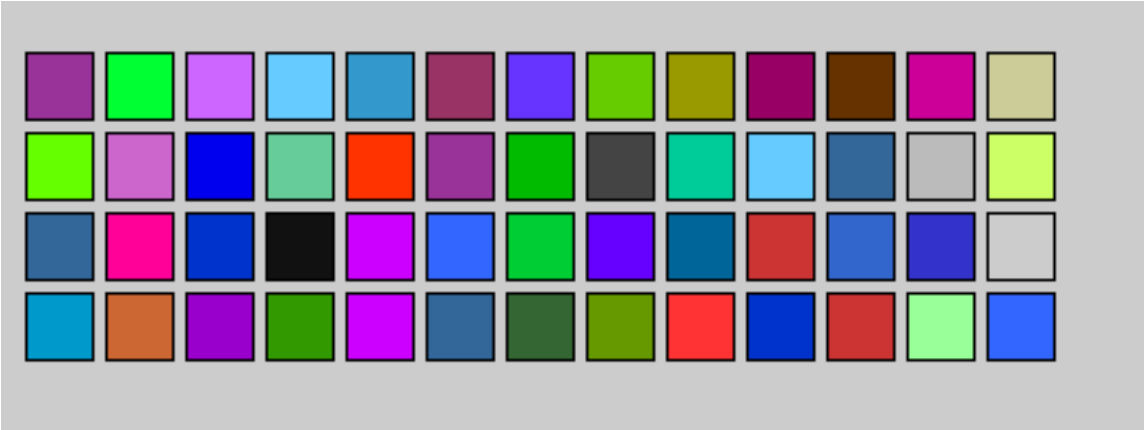
```
import java.awt.*;

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);
      }
  }
}
```

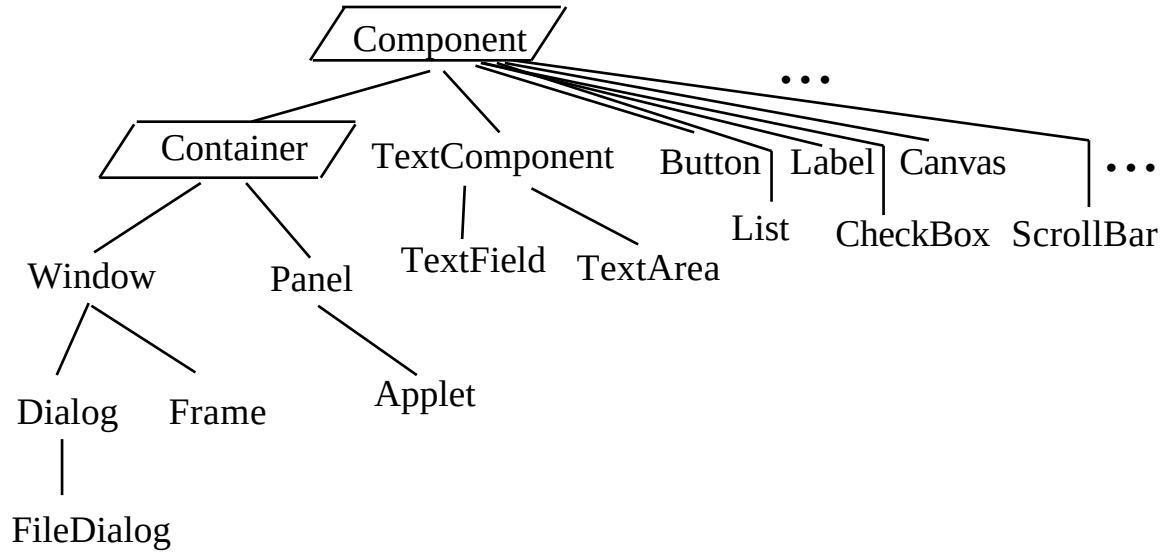


On peut utiliser les constantes de la classe `Color` comme `Color.green`, `Color.pink`, ...

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)

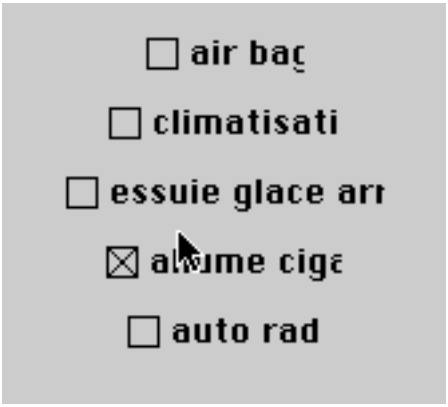
les cases à cocher (suite)

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 :



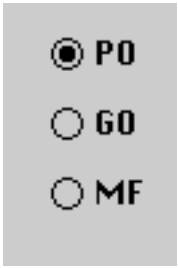
Composants graphiques de base (suite)

cases à option

On construit les mêmes composants que ci-dessus et on les met dans une boîte qui les gèrent 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 chaine 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()`.

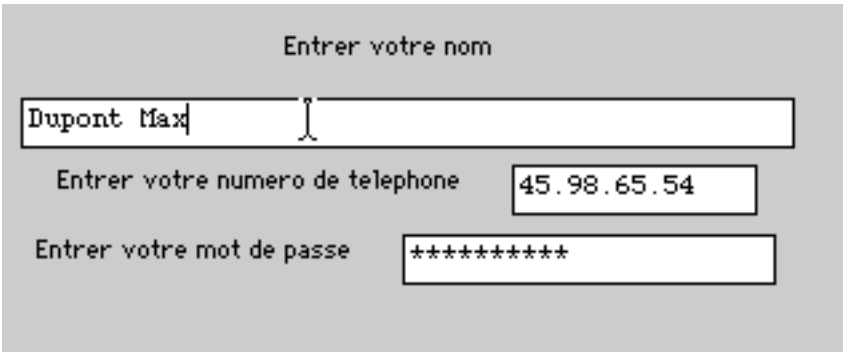
Composants graphiques de base (fin)

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 :



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 applets), `GridLayout`, `BorderLayout` (politique par défaut dans les applications).

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

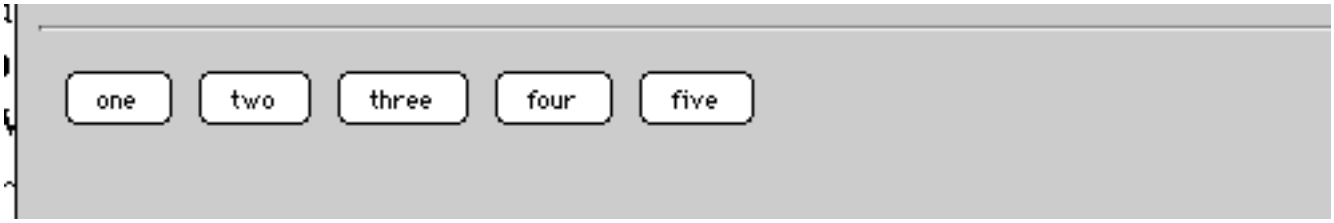
placement FlowLayout

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));
```

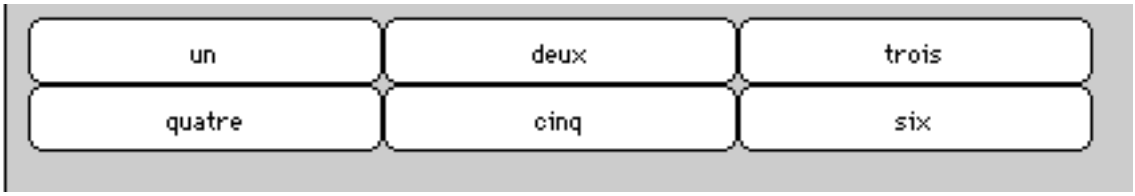
Les politiques de placement

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.

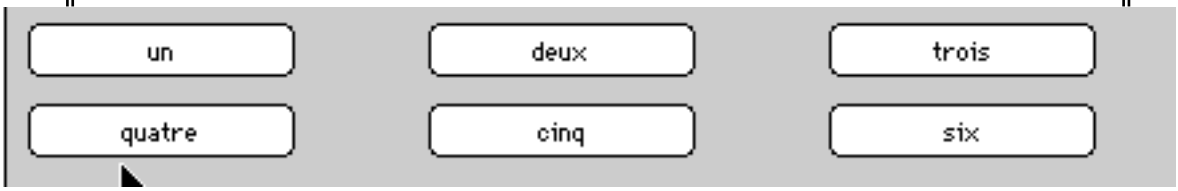
```
setLayout(new GridLayout(2,3));
add(new Button("un"));
add(new Button("deux"));
add(new Button("trois"));
add(new Button("quatre"));
add(new Button("cinq"));
add(new Button("six"));
```

donne



on peut aussi indiquer les espacements par :

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



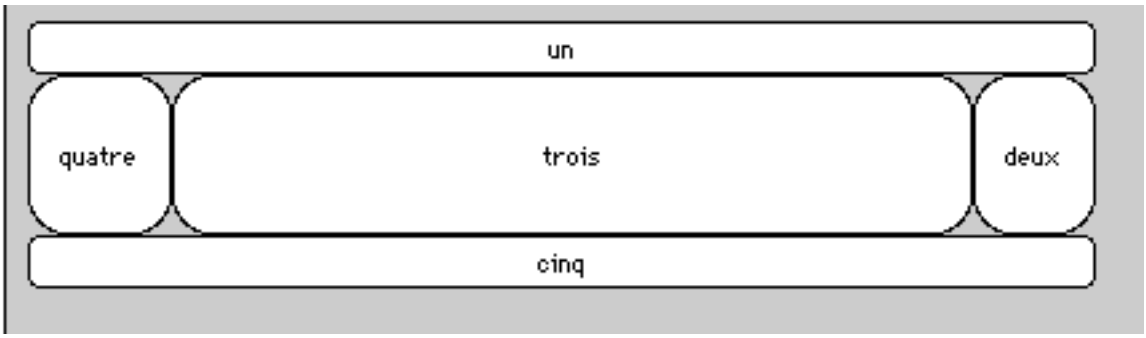
Les politiques de placement

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 :



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.

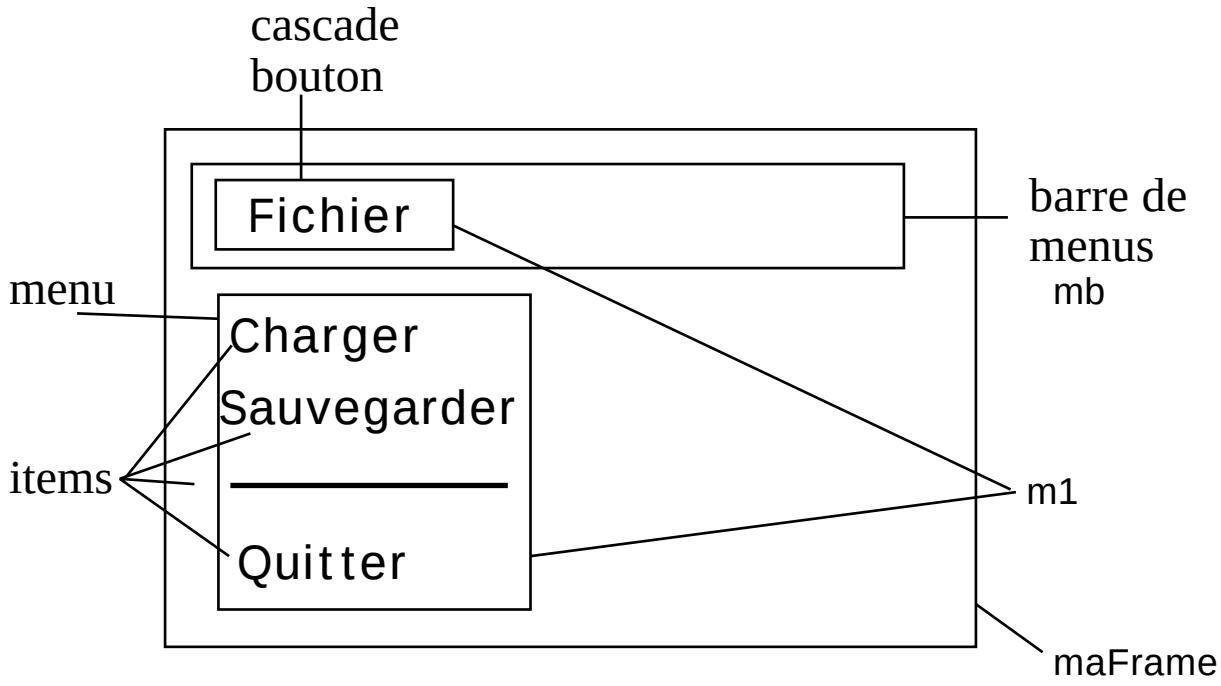
Si ce conteneur est "géré par un `FlowLayout` ou un `GridLayout`", cette méthode doit avoir 1 seul argument : l'objet à ajouter.

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

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("Quitte"));

mb.add(m1);

maFrame.setMenuBar(mb);
```



Les méthodes graphiques

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

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) efface le composant (le redessine avec sa couleur de fond), puis appelle paint(Graphics g).

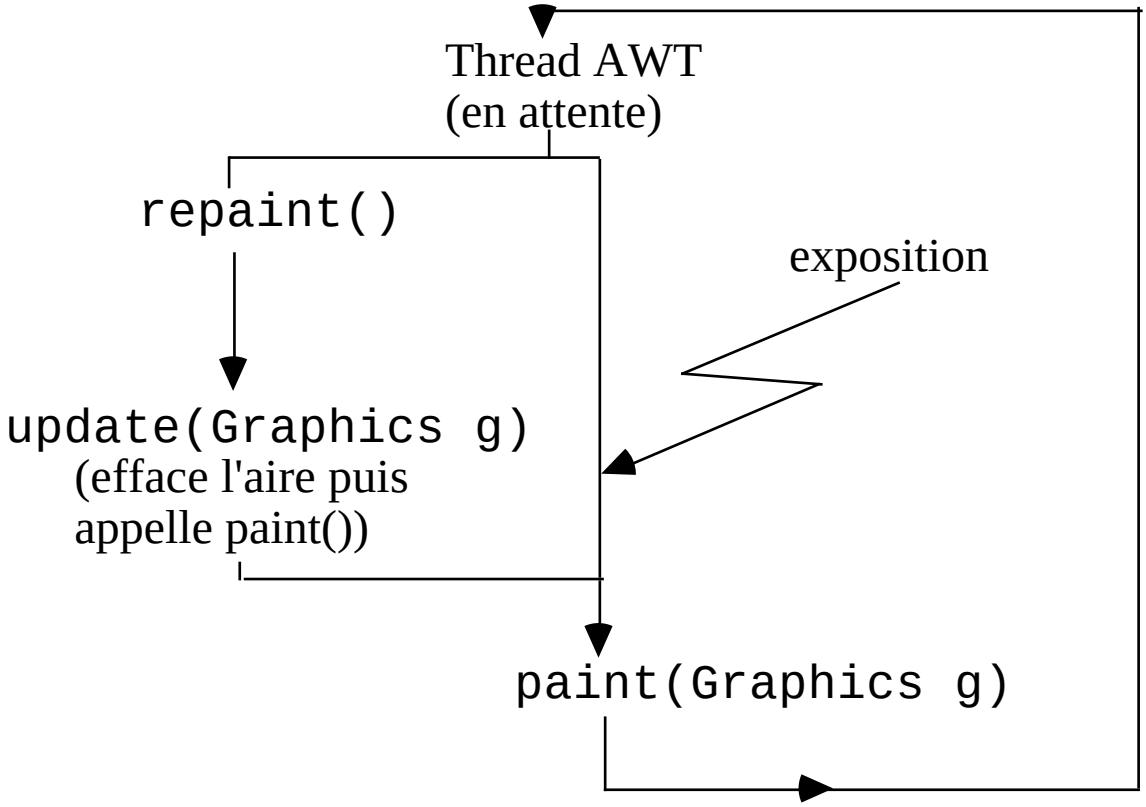
D'ailleurs update(Graphics g) de la classe Component est :

```
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 :

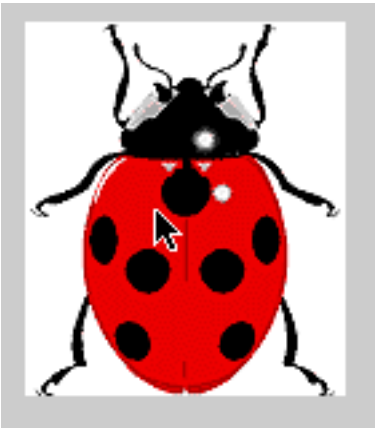
```
public void update(Graphics g) {
        paint(g);
}
```

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);
    }
}
```



Les événements en Java 1.1

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'ont sait quelles méthodes implémenter. Ces méthodes sont de la forme `public void XXXYYY(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 avoir alors avoir les renseignements de la situation lors de la production de cet événement entre autre le composant graphique qui a été utilisé par l'utilisateur 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();
new WindowCloser(maFrame);
```

où, si on veut terminer complètement l'application, écrire :

```
new WindowCloser(maFrame, true);
```

avec la classe `WindowCloser` ci dessous :

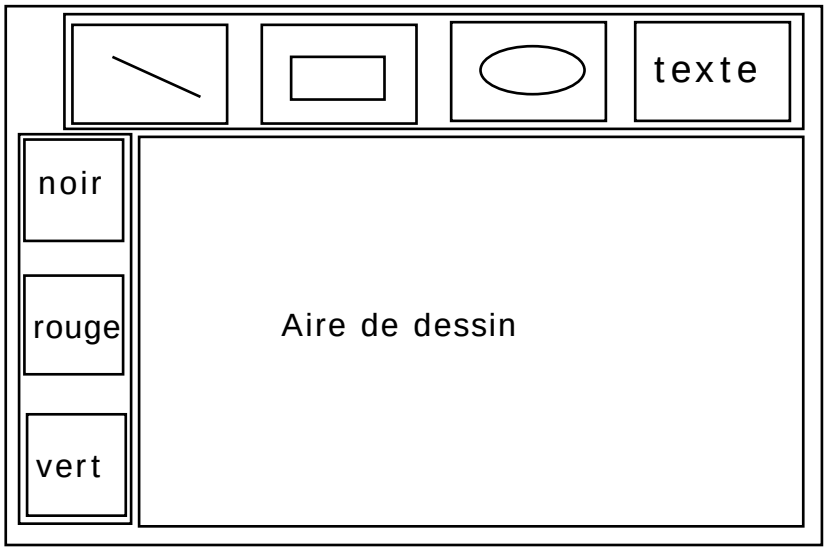
```
import java.awt.*;
import java.awt.event.*;

public class WindowCloser implements
WindowListener {

    private boolean exitOnClose;

    public WindowCloser(Window w) {
       this(w, false);
    }

    public WindowCloser(Window w, boolean
exitOnClose) {
       this.exitOnClose = exitOnClose;
       w.addWindowListener(this);
    }
```

```
/* appelé avant que la fenêtre soit fermé. On peut
ainsi outrepasser cette fermeture, faire des
sauvegardes, ... Ce n'est pas le cas ici.
*/
    public void windowClosing(WindowEvent e) {
        if (exitOnClose) {
            System.exit(0);
        }
        e.getWindow().dispose();
    }

/* invoqué après que la fenêtre ait été fermé */
    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) {
}

}
```

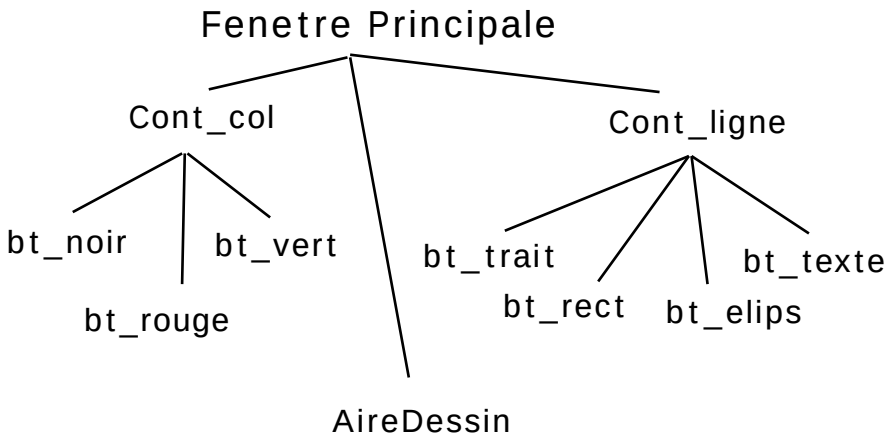
Appendice : Le mécanisme des événements en Java 1.0

Gestion des événements en Java 1.0

L'interface graphique

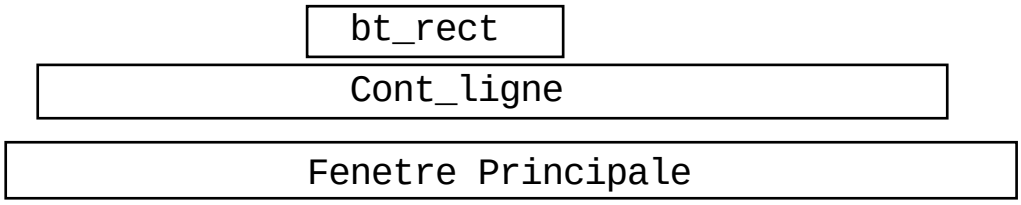


correspond à l'arborescence :

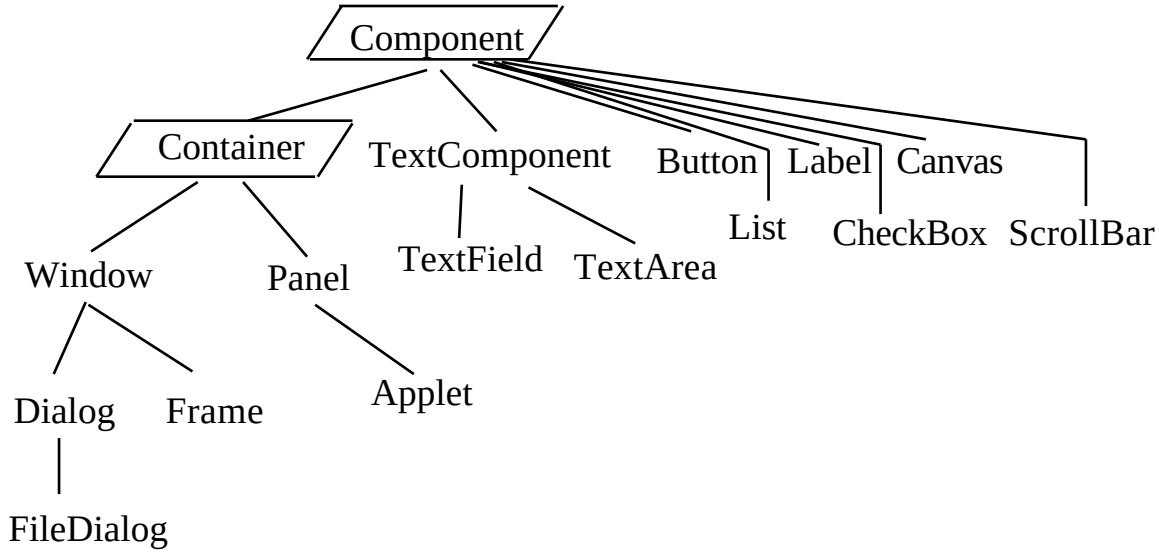


La gestion des événements (suite)

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

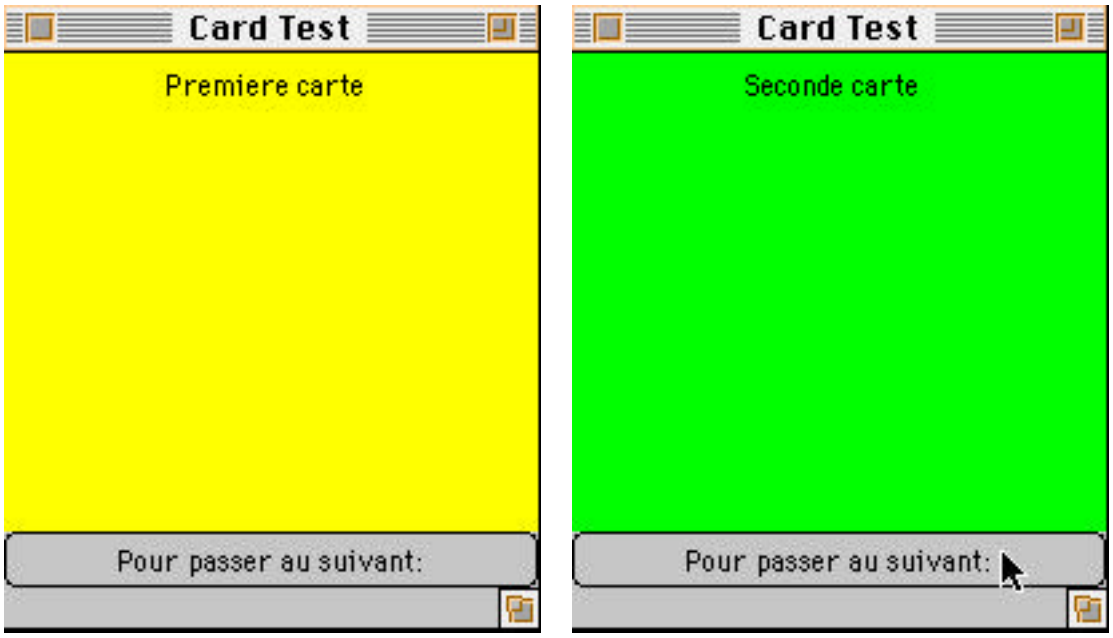
```
import java.awt.*;

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

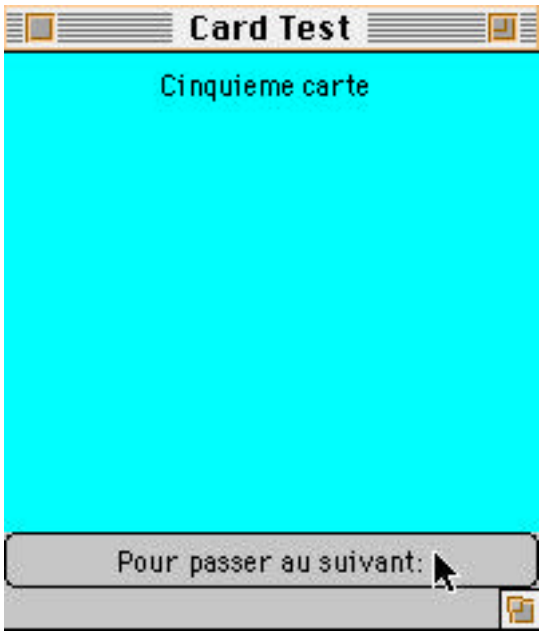
Un panneau menant une politique `CardLayout` ne fait apparaître qu'un seul de ses contenants à la fois.
En général on crée un tel panneau `panCard`, on met les composants un à un dans ce panneau, puis on accède à ces éléments en lançant les méthodes `first()`, `last()`, `next()`, `previous()` sur cette 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);
      }
}
```

Programmation graphique avancée et animations (suite)

Une animation

Inspirée de celle écrite par Arthur van Hoff qui se trouve à se trouve :
http://www.javaworld.com/jw-03-1996/animation/Example7Applet.html
écrite par Arthur van Hoff extrait de son article sur la programmation d'animations en Java disponible à :
http://www.javaworld.com/jw-03-1996/animation
J'ai ajouté le chargement par le MediaTracker, une réécriture de update() et paint().
On dispose des 2 images :

world.gif: 

car.gif: 

2 voitures avancent de droite à gauche sur l'image du monde.

Une animation (A. V. Hoff)

le fichier html contient :

```
<applet code=Example7Applet.class width=200 height=200>
<param name=fps value=20>
</applet>
```

applet d'animation

```
import java.awt.*;

public
class Example7Applet extends
java.applet.Applet implements Runnable {
    int frame;
    int delay;
    Thread animator;

    Dimension offDimension;
    Image offImage;
    Graphics offGraphics;

    Image world;
    Image car;

    /**
     * Initialisation de l'applet et calcul du delai
     * entre "trames".
     */
    public void init() {
        String str = getParameter("fps");
     int fps = (str != null) ? Integer.parseInt(str) : 10;
        delay = (fps > 0) ? (1000 / fps) : 100;

        tracker = new MediaTracker(this);
        world = getImage(getCodeBase(),
"world.gif");
        car = getImage(getCodeBase(), "car.gif");
        tracker.addImage(world, 0);
        tracker.addImage(car, 0);
    }
```

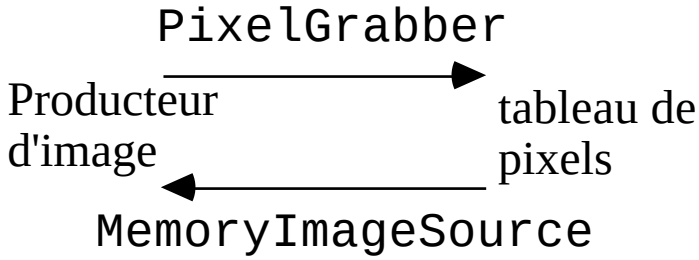
```
    /**
     * La methode start() de l'applet. On crée la
     * thread d'animation et on la lance.
     */

    public void start() {
        animator = new Thread(this);
        animator.start();
    }

    /**
     * Le corps de la thread.
     */
    public void run() {
        // stocker la date de lancement
        long tm = System.currentTimeMillis();
        while (Thread.currentThread() == animator) {
            // lance l'affichage de l'animation
            repaint();

            // Delai d'attente ajuste pour avoir la
            // meme attente entre chaque trame.
            try {
                tm += delay;
                Thread.sleep(Math.max(0, tm -
System.currentTimeMillis()));
            } catch (InterruptedException e) {
                break;
            }

            // numero de trame incremente pour
            // pouvoir afficher la trame
            // suivante.
            frame++;
        }
    }
```

```
    /**
     * La methode stop() de l' applet.
     * Elle arrete la thread et desalloue
     * les entites necessaires au double buffering.
     */
    public void stop() {
        animator = null;
        offImage = null;
        offGraphics = null;
    }

    /** dessine la trame courante. */
    public void paint(Graphics g) {
        Dimension d = size();

// Cree le double buffering si ce n'est pas deja
fait.
        if ((offGraphics == null)
         || (d.width != offDimension.width)
         || (d.height != offDimension.height)) {
           offDimension = d;
           offImage = createImage(d.width, d.height);
           offGraphics = offImage.getGraphics();
       }

        // efface l'image precedente
        offGraphics.setColor(getBackground());
        offGraphics.fillRect(0, 0, d.width, d.height);
        offGraphics.setColor(Color.black);

        // prepare le dessin de la bonne trame
        paintFrame(offGraphics);

        // affiche la bonne trame a l'écran
        g.drawImage(offImage, 0, 0, null);
    }
```

```
    public void update(Graphics g) {
        paint(g);
    }
    /** la creation de la trame :
      * utilise le double buffering et le MediaTracker
      */
    public void paintFrame(Graphics g) {
        // Ne faire l'affichage que lorsque
        // toutes les images ont été chargées
        // au besoin provoquer le chargement
        // des images par status(..., true);
        if (tracker.statusID(0, true) ==
MediaTracker.COMPLETE) {
        Dimension d = size();
        int w = world.getWidth(this);
        int h = world.getHeight(this);
            g.drawImage(world, (d.width - w)/2,
(d.height - h)/2, this);
        w = car.getWidth(this);
        h = car.getHeight(this);
            w += d.width;
            //dessine la premiere voiture qui avance de
             // a la fois 5 pixels de droite à gauche
            g.drawImage(car, d.width - ((frame * 5) %
w), (d.height - h)/3, this);
            //dessine la seconde voiture :
            // elle avance de 7 pixels de droite à
gauche
            g.drawImage(car, d.width - ((frame * 7) %
w), (d.height - h)/2, this);
        }
        else { // dans le cas où les images n'ont
        // pas encore été chargées
g.drawString("Images en cours de chargement",
50, 50);
        }
    }
}
```

Manipulations d'images

Java propose des classes permettant de manipuler les images. Plus précisément de faire correspondre des tableaux de pixels à des images. Ce sont les classes `PixelGrabber` et `MemoryImageSource`.



Le tableau de pixels est un tableau d'entiers contenant les valeurs RGB des pixels.

La classe `PixelGrabber` implémente l'interface `ImageConsumer`.

La classe `MemoryImageSource` implémente l'interface `ImageProducer`.

Ces classes font partie du paquetage `java.awt.image`.

La classe PixelGrabber principales méthodes

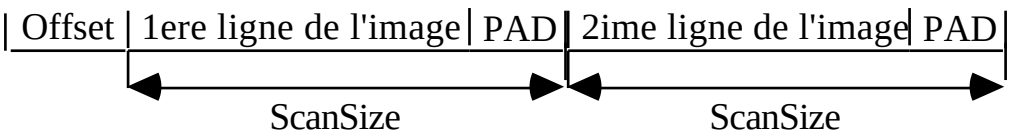
`public PixelGrabber(Image img, int x, int y, int w, int h, int tabpixels[ ], int offset, int scansize)`

Les pixels de la sous image `x`, `y`, `w`, `h` de `img` seront stockés dans le tableau `tabpixels` de sorte que le pixel(i, j) ait sa valeur stockée à

`tabpixels[(j-y)*scansize + (i-x) + offset]`

`offset` donne l'indice où commence véritablement les données pixels.

`scansize` indique la taille d'une ligne de l'image : c'est souvent la largeur de l'image.



`public boolean grabPixels() throws InterruptedException`

C'est la méthode qui effectue véritablement le chargement des pixels de l'image vers le tableau. L'exception `InterruptedException` est levée si une autre thread interrompt celle-ci.

La classe MemoryImageSource principales méthodes

`public MemoryImageSource(int w, int h, int tabpixels[ ], int offset, int scansize)`

construit un producteur d'image à partir du tableau de pixels `tabpixels`.

On obtient un objet de la classe `Image` à partir d'un producteur d'image grâce à la méthode
`Image createImage(MemoryImageSource)`
de la classe `Component`.

Un exemple

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

public class TraiteImage extends Applet {
	private img, imgtraite;

	public void init() {
	img = getImage(getDocumentBase(),
"belImage.gif");
	int w = img.getWidth(this);
	int h = img.getHeight(this);
	int tabpix[ ] = new int [w * h];
	PixelGrabber grab = new PixelGrabber(
		img, 0, 0, w, h, tabpix, 0, w);
	try {
		grab.grabPixels();
	} catch (InterruptedException e) {
		System.out.println("Pb dans
grabPixels()");
	}
// traitement de l'image à l'aide des
// elements du tableau

	MemoryImageSource mem = new
MemoryImageSource(w, h, tabpix, 0, w);
	imgtraite = createImage(mem);
	}
}
```

Les images filtrées

Java propose un ensemble de classes (dans le paquetage `java.awt.image`) pour faire des "filtres" sur les images. Ce sont les classes :
`FilteredImageSource`, `ImageFilter`, `RGBImageFilter`

`public class FilteredImageSource implements ImageProducer`
permet de construire une nouvelle image à partir d'une image existante et d'un filtre.

`public class ImageFilter implements ImageConsumer, Cloneable`
permet de construire des filtres pour un producteur d'image vers un consommateur d'image. Les filtres seront des objets de classes dérivées de cette classe.

`public abstract class RGBImageFilter extends ImageFilter`
permet de construire facilement des filtres à l'aide de valeur RGB. Cette classe abstraite contient plusieurs méthodes possédant une définition et une seule méthode abstraite :
`public abstract int filterRGB(int x, int y, int rgb)`.

La classe FilteredImageSource

`public FilteredImageSource (ImageProducer source, ImageFilter filtre)`

construit une image filtrée à partir de le l'objet `ImageProducer source` à l'aide du filtre `filtre`.

remarque :

on obtient l'`ImageProducer` d'un objet `img` de la classe `Image` à l'aide de : `img.getSource()`.

La classe RGBImageFilter

principaux champs

`protected boolean canFilterIndexColorModel`

Si égal à `true`, indique que la valeur retournée par la méthode `filterRGB()` est indépendante de la position x, y du pixel dans l'image.

Exemple

```
// fortement inspiré de Java in a Nutshell
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class GrayButton extends Applet {
	Image img, imagegrise;
	int w, h;

	public void init() {
		img = getImage(getDocumentBase(),
"images/ladybug.gif");
		w = img.getWidth(this);
		h = img.getHeight(this);
		ImageFilter filtre = new FiltreGris();
		ImageProducer prod = new
FilteredImageSource(img.getSource(), filtre );
		imagegrise = createImage(prod);
	}

	public void update(Graphics g) {
		g.drawImage(img, 0, 0, this);
	}

	public boolean mouseDown(Event e, int x,
int y) {
		Graphics legc = this.getGraphics();
		Dimension taileApplet = this.size();
		legc.clearRect(0, 0, taileApplet.width,
taileApplet.height);
		legc.drawImage(imagegrise, 0, 0, this);
		return true;
	}
```

```
	public boolean mouseUp(Event e, int x, int y)
{
		update(this.getGraphics());
		return true;
	}
}

class FiltreGris extends RGBImageFilter {
	public FiltreGris () {
canFilterIndexColorModel = true; }
	public int filterRGB(int x, int y, int rgb) {
		int a = rgb & 0xff000000;
		int r = ((rgb & 0xff0000) + 0xff0000)/2;
		int g = ((rgb & 0x00ff00) + 0x00ff00)/2;
		int b = ((rgb & 0x0000ff) + 0x0000ff)/2;
		return a | r | g | b ;
	}
}
```

Les boutons images

Le code proposé est en Java 1.0. On veut avoir :

```
public class AppletBoutonImage extends Applet {
       BoutonImage fauxBt1, fauxBt2;

      public void init() {
                    fauxBt1 = new BoutonImage( this,
"ladybug", "ladybug" );
                    fauxBt2 = new BoutonImage( this,
"europe",  "europe" );
                  add( fauxBt1 );
                  add( fauxBt2 );
     }


       public boolean action(Event evt, Object arg )
{
                   if ( evt.target instanceof Button )    {
                         if ( arg.equals("europe") )
              System.out.println( "ca vient de la carte
d'europe" );
                          if ( arg.equals("ladybug") )
              System.out.println( "ca vient de l'image
de la coccinelle" );
                         return true;
                 }
                  return false;
            }
}
```

Les boutons images (suite)

pour cela on construit :

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

class BoutonImage extends Canvas  {
          Image                 im, gray;
          Applet              appO;
          int                   lg, ht;
          Event               evt;
          Button              Bt;
          String              argument;
           MediaTracker    tracker;

BoutonImage(Applet appletOrig, String image,
String arg){
     appO          = appletOrig;
     argument      = arg;
      tracker = new MediaTracker( this );

     im =
appO.getImage(appO.getDocumentBase(),"ima
ges/"+image+".gif");
      tracker.addImage( im, 1 );
     try    {
            tracker.waitForID( 1 );
     }      catch( InterruptedException e ) {
       System.out.println("Erreur dans le
tracker.waitForID( 1 )");
       }
```

```
    if ( tracker.isErrorID ( 1 ) )
            System.out.println("Erreur au
chargement de l'image");
      int resulMT;
      resulMT = tracker.statusID(1, false);
      if ((resulMT & MediaTracker.COMPLETE) !=
0)
            System.out.println("SUPER le
chargement de l'image s'est BIEN effectue");

       int l = im.getWidth ( this );
      int h = im.getHeight( this );
      resize( l, h );

      ImageFilter f = new FiltreGris();
      ImageProducer producer = new
FilteredImageSource(im.getSource(), f);
      gray = this.createImage(producer);
      Bt = new Button(argument );
}

public void paint(Graphics g) {
  g.drawImage(im, 2, 2, this);
}

public boolean mouseDown(Event e, int x, int y) {
  Graphics g      = this.getGraphics();
  Dimension d = this.size();
  g.clearRect(0, 0, d.width, d.height);
  g.drawImage(gray, 2, 2, this);
  evt = new Event(Bt, Event.ACTION_EVENT,
argument );
   deliverEvent( evt );
   return true;
}
```

```
    public boolean mouseUp(Event e, int x, int y) {
        update(this.getGraphics());
        return true;
    }
}


class FiltreGris extends RGBImageFilter {
    public FiltreGris() {
        canFilterIndexColorModel = true;
    }

    public int filterRGB(int x, int y, int rgb) {
      int a =   rgb & 0xff000000;
      int r = ((rgb & 0xff0000) + 0xff0000)/2;
      int g = ((rgb & 0x00ff00) + 0x00ff00)/2;
      int b = ((rgb & 0x0000ff) + 0x0000ff)/2;
      return a | r | g | b;
    }
}
```

On a associé à notre image, un `Button` Java nécessaire lorsqu'on veut gérer les événements par le constructeur `Event(Object target, int id, Object arg)` qui créé un événement Java où `target` sera le composant graphique initiateur, `id` le type de l'événement et `arg` l'argument (qu'on peut récupérer dans `action(...)`)

Observer/Observable

La classe `Observable` et l'interface `Observer` implantent le modèle MVC ou plus précisement Model-View de Smalltalk.

A chaque objet de `Observable` est associé une liste d'objets d'une classe implémentant `Observer`. Ces objets sont informés de changement lorsque l'`Observable` lance la méthode `notifyObservers()`(après avoir indiqué un tel changement par `setChanged();`). Les observers associés lanceront leur méthode `update()`

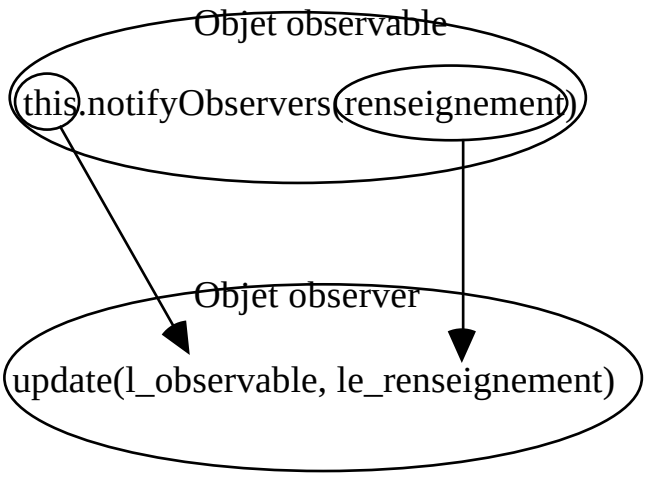
Observer

`java.util.Observer` est une interface `public interface Observer` qui contient l'unique signature :
`public abstract void update(Observable o, Object arg)` méthode appelée lorsque l'instance appartient à la liste des observers de l'objet `o`.
`arg` est un paramètre permettant de passer une information de l'`Observable` à ses `Observer` associés.

Observable

`java.util.Observable` est une classe
Une notification de changement dans l'instance est passée aux `Observer` associés en lançant la méthode `notifyObservers()` (qui lancera la méthode `update()` sur chaque `Observer` associé).

On a donc :



Observable (suite)

Principales méthodes

Un seul constructeur : `Observable()`

`addObserver(Observer)` ajoute un observer à la liste des observers de l'instance.
`countObservers()` retourne le nombre d'observers.
`deleteObserver(Observer)` enlève de l'instance l'observer référencé par l'argument.
`deleteObservers()` enlève tous les observers de l'instance.
`hasChanged()` retourne `true` si un changement est survenu dans l'instance.
`notifyObservers()` indique aux observers qu'un changement est survenu.
`notifyObservers(Object)` idem que la méthode précédente où on passe une donnée référencée par l'argument aux observers.
`setChanged()` positionne le drapeau de l'instance indiquant qu'un changement s'est produit.
`clearChanged()` marque l'instance comme inchangée.

rappel important
Il faut utiliser `setChanged()` avant `notifyObservers()` pour prendre en compte les changements.

Observer/Observable (suite)

Un exemple

```
import java.util.*;

public class ListeDesConnectes extends Observable {
        private Hashtable LesConnectes = new Hashtable();

       public void login(String nom, String motPasse) throws
BadUserException
      {
         // test du mot de passe
         if (!MotDePasseValid(nom, motPasse))
              throw new MotDePasseException(nom);

              // on a besoin de 2 arguments pour put() de l'objet
              // LesConnectes de la classe java.util.Hashtable
              Utilisateur utilaux = new Utilisateur(nom);
              LesConnectes.put(nom, utilaux);

          // notifyObservers() n'informe les Observers
          // que si il y a eu des changements et il y a
          // des changements que si on les fait
          // explicitement par setChanged()
              setChanged();
              notifyObservers(utilaux);
      }

       public void logout(Utilisateur unUtil) {
            // utilisation de remove() de Hashtable
              LesConnectes.remove(unUtil.getNom());
             setChanged();
              notifyObservers(unUtil);
      }
       public Hashtable getLesConnectes() {
              return LesConnectes;
      }
// ...
}
```

```
public class Controleur implements Observer {
        ListeDesConnectes liste;

    public Controleur(ListeDesConnectes tousLesConnectes) {
                liste = tousLesConnectes;
                // au moment de la création (i.e. dans le constructeur),
                // on prévient l'observable qu'on est la pour l'observer
                // et qu'il nous informe de ses changements
                liste.addObserver(this);
}

     public void update(Observable lesConnectes, Object
renseignements)
        {
          Utilisateur unUtil = (Utilisateur) renseignements;
            if (liste.getLesConnectes().contains(unUtil))
                        ajouterUtilisateur(unUtil);
              else
                        enleverUtilisateur(unUtil);
        }
}
```

Animation : Neko le chat

C'est une animation graphique écrite à l'origine pour macintosh par Kenjo Gotoh en 1989. Elle montre un petit chat (Neko en japonais), qui court de gauche à droite, s'arrête, baille, se gratte l'oreille, dort un moment puis sort en courant vers la droite. On utilise pour cela les images suivantes :

Le code de l'animation Neko

```
import java.awt.Graphics;
import java.awt.Image;
import java.awt.Color;

public class Neko extends java.applet.Applet
implements Runnable {

  Image nekopics[] = new Image[9];
  String nekosrc[] = { "right1.gif", "right2.gif",
"stop.gif","yawn.gif", "scratch1.gif", "scratch2.gif",
"sleep1.gif", "sleep2.gif", "awake.gif" };
  Image currentimg;
  Thread runner;
  int xpos;
  int ypos = 50;

  public void start() {
    if (runner == null) {
      runner = new Thread(this);
      runner.start();
    }
  }

  public void stop() {
    if (runner != null) {
      runner.stop();
      runner = null;
    }
  }
```

```
  public void run() {
    // chargement des images
    for (int i=0; i < nekopics.length; i++) {
      nekopics[i] = getImage(getCodeBase(),
"images/" + nekosrc[i]);
    }
    setBackground(Color.white);
     int milieu = (this.size().width) / 2;
     System.out.println("milieu = " + milieu);

    // court de gauche a droite
    flipflop(0, milieu, 150, 1, nekopics[0],
nekopics[1]);

    // s'arrete
    flipflop(milieu, milieu, 1000, 1, nekopics[2],
nekopics[2]);

    // baille
    flipflop(milieu, milieu, 1000, 1, nekopics[3],
nekopics[3]);

    // se gratte 4 fois
    flipflop(milieu, milieu, 150, 4, nekopics[4],
nekopics[5]);

    // dort i.e. ronfle 5 fois
    flipflop(milieu, milieu, 250, 5, nekopics[6],
nekopics[7]);

    // se reveille
    flipflop(milieu, milieu, 500, 1, nekopics[8],
nekopics[8]);

    // et part
    flipflop(milieu, this.size().width + 10, 150, 1,
nekopics[0], nekopics[1]);
  }
```

```
  void flipflop(int debut, int end, int timer, int
nbFois, Image img1, Image img2) {
    for (int j = 0; j < nbFois; j++) {
    for (int i = debut; i <= end; i+=10) {
      this.xpos = i;
      // swap images
      if (currentimg == img1)
        currentimg = img2;
      else if (currentimg == img2)
        currentimg = img1;
      else currentimg = img1;

      repaint();
      pause(timer);
     }
    }
  }

  void pause(int time) {
    try { Thread.sleep(time); }
    catch (InterruptedException e) { }
  }

  public void paint(Graphics g) {
    g.drawImage(currentimg, xpos, ypos,this);
  }
}
```

Les sons dans les applets

Java propose la classe `java.applet.AudioClip` permettant de manipuler les sons.
Pour l'instant les seuls fichiers sons disponibles (jusqu'à Java 1.2 et Java media Framework) sont les `.au`, un format développé par Sun et jouable sur diverses plateformes.
On peut charger facilement des sons dans une applet grâce aux méthodes
`public AudioClip getAudioClip(URL url)`
ou
`public AudioClip getAudioClip(URL url, String nom)`
de la classe `Applet`.

Après avoir récupérer un objet de la classe `AudioClip` on peut utiliser les 3 méthodes de cette classe : `play()`, `loop()`, `stop()`.
Il est conseillé de lancer la méthode `stop()` de la classe `AudioClip` dans la méthode `stop()` de la classe `Applet`.

Programme de Sons

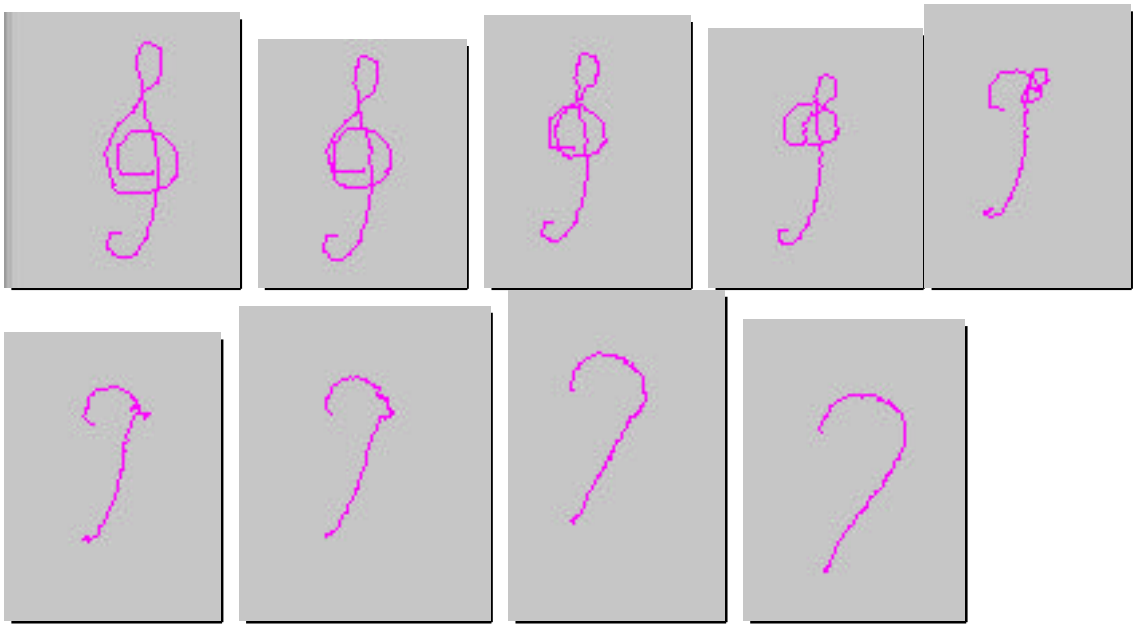
```
import java.awt.Graphics;
import java.applet.AudioClip;

public class AudioLoop extends
java.applet.Applet implements Runnable {

  AudioClip bgsound;
  AudioClip beep;
  Thread runner;

  public void start() {
    if (runner == null) {
      runner = new Thread(this);
      runner.start();
    }
  }

  public void stop() {
    if (runner != null) {
      if (bgsound != null)
        bgsound.stop();
      runner.stop();
      runner = null;
    }
  }
```

```
  public void init() {
    bgsound = getAudioClip(getCodeBase(),
"audio/loop.au");
    beep = getAudioClip(getCodeBase(),
"audio/beep.au");
  }

  public void run() {
    if (bgsound != null) bgsound.loop();
    while (runner != null) {
      try { Thread.sleep(5000); }
      catch (InterruptedException e) { }
      if (bgsound != null) beep.play();
    }
  }

  public void paint(Graphics g) {
    g.drawString("Execution de musique....", 10,
10);
  }
}
```

Morphing 2D en Java

Une applet écrite par Pat Niemeyer (pat@pat.net) utilisable à : http://www.ooi.com/tween/editor.html

On fait 2 dessins et le programme passe de l'un à l'autre par "morphing".



"... feel free to use it for your class but please don't redistribute it"

code morphing P. Niemeyer

L'applet utilise essentiellement 2 classes dans le paquetage `tween` : la classe `Editor` et la classe `Canvas`. L'applet présente tout d'abord un éditeur de dessin :

dont une partie du code est :

```
package tween;
•••
class Editor extends Panel {
        tween.Canvas canvas;
        Graphics canvasGr;
        int xpos, ypos, oxpos, oypos;
        private int [][] track = new int [2048][2];
        private int trackEnd = 0;
        private Vector tracks = new Vector();
        static boolean standalone;

        Editor() {
                •••
                add( "Center", canvas = new tween.Canvas() );
                Panel p = •••
                p.add( new Button("Clear") );
                p.add( new Button("Tween") );
                add( "South", p );
        }
```

```
final public boolean handleEvent( Event e ) {
        int x = e.x, y = e.y;
        if ( e.id == Event.MOUSE_DRAG ) {
                xpos = x; ypos = y;
                addPoint( x, y );
                if ( canvasGr == null )
                        canvasGr = canvas.getGraphics();
                canvasGr.setColor( Color.red );
                canvasGr.drawLine( oxpos, oypos,
oxpos=xpos, oypos=ypos );
                return true;
        } else
        if ( e.id == Event.MOUSE_DOWN ) {
                oxpos = x; oypos = y;
                trackEnd = 0;
                addPoint( x, y );
                return true;
        } else
        if ( e.id == Event.MOUSE_UP ) {
                int l;
                int [][] tr = new int [ trackEnd ][2];
                for ( int i=0; i< tr.length; i++) {
                        tr[i][0] = track[i][0];
                        tr[i][1] = track[i][1];
                }
                tracks.addElement( tr );
                return true;
        }
        return true;
    }
    public boolean action( Event e, Object o ) {
        ••• if ( o.equals("Tween") ) {
                canvas.setTracks( tracks );
                canvas.startTweening(); •••
        }
        return true;
    }

    private void addPoint( int x, int y ) {
        try {
                track[trackEnd][0] = x;
                track[trackEnd][1] = y;
                trackEnd++;
        } catch ( ArrayIndexOutOfBoundsException e2 ) {
                System.out.println("too many points!!!");
                trackEnd = 0;
        }
    }
}
```

morphing P. Niemeyer (suite)

L'éditeur est un code classique de "rubber band" avec les 3 cas à traiter de bouton souris : appui, déplacer bouton enfoncé, relachement. À chaque manipulation, le code remplit un tableau `track` (indice `trackEnd` incrémenté) qui va "contenir la courbe tracé par l'utilisateur".
Lorsque le tracé est fini ce tableau est alors mis dans le vecteur `tracks` qui constitue alors le premier élément du vecteur. Le tracé suivant sera le second élément du vecteur, (etc. car on peut mettre plusieurs tracés).
Un tracé est contitué d'un ensemble de points qui ont eux même 2 coordonnées d'où la declaration de tableau a deux indices de la forme : `track[numeroDuPoint][0 pour x, 1 pour y]`

morphing P. Niemeyer (suite)

La seconde classe importante est `tween.Canvas` :

```
package tween;

import java.awt.*;
import java.util.Vector;
import java.io.*;

class Canvas extends java.awt.Canvas implements Runnable
{
	private Vector tracks = new Vector();
	private Thread runner;
	private Image drawImg;
	private Graphics drawGr;
	private boolean loop;

	void setTracks( Vector tracks ) {
		if ( runner != null )
			stopTweening();
		this.tracks = tracks;
	}
	synchronized public void startTweening( ) {
		if ( runner != null )
			stopTweening();
		if ( tracks == null || tracks.size() == 0 )
			return;
		runner = new Thread( this );
		runner.setPriority( runner.getPriority() + 1 );
		this.loop = false;
		runner.start();
	}
	synchronized public void stopTweening() {
		if ( runner == null )
			return;
		runner.stop();
		runner = null;
	}
	public void run() {
		do {
			tweenTracks();
		} while ( loop );
	}
```

```
        private void tweenTracks() {
                int n = tracks.size();
                for (int i=0; i< n-1; i++)  {
                        tweenTrack( (int [][])tracks.elementAt(i),
                                (int [][])tracks.elementAt(i+1) );
                }
                // draw the final track precisely...
                // (we've tweened "up to" it)
                clearGr();
                drawTrack( (int [][])tracks.elementAt(n-1),
Color.magenta );
        }

        private void tweenTrack(
                int [][] fromTrack, int [][] toTrack ) {
                int tweens = averageDistance( fromTrack, toTrack
)/2;
                if ( tweens < 1 )
                        tweens = 1;
                for ( int tweenNo=0; tweenNo < tweens;
tweenNo++) {
                        int len = (int)( fromTrack.length
                                + 1.0*(toTrack.length -
fromTrack.length)/tweens * tweenNo );
                        int [][] tweenTrack = new int [len][2];

                        for ( int i = 0; i < tweenTrack.length; i++ ) {
                                int from =
(int)(1.0*fromTrack.length/tweenTrack.length * i);
                                int to =
(int)(1.0*toTrack.length/tweenTrack.length * i);
                                int x1 = fromTrack[from][0];
                                int x2 = toTrack[to][0];
                                int y1 = fromTrack[from][1];
                                int y2 = toTrack[to][1];

                                tweenTrack[i][0] = (int)(x1 + 1.0*(x2-
x1)/tweens * tweenNo);
                                tweenTrack[i][1] = (int)(y1 + 1.0*(y2-
y1)/tweens * tweenNo);
                        }

                        clearGr();
                        drawTrack( tweenTrack, Color.magenta );
                        try {
                                Thread.sleep( 1000/24 );
                        } catch ( Exception e ) { }
                }
        }
```

```
        /*
                Sample 1/10 of the (average) number of points to
calculate an
                average overall distance
        */
        private int averageDistance( int [][] track1, int [][] track2 )
{
                int averages = (track1.length + track2.length)/2/10;
                if ( averages < 3 )
                        averages = 3;
                int t = 0;
                for ( int i=0; i< averages; i++ ) {
                        int [] p1 = track1[ (int)( 1.0 * track1.length /
averages * i ) ];
                        int [] p2 = track2[ (int)( 1.0 * track2.length /
averages * i ) ];
                        int dx = p2[0] - p1[0];
                        int dy = p2[1] - p1[1];
                        t += (int)Math.sqrt( dx*dx + dy*dy );
                }
                return t/averages;
        }

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

        public void paint( Graphics g ) {
                if ( drawImg == null )
                        return;
                g.drawImage(drawImg, 0, 0, null);
        }

        public void drawTrack(int [][] track, Color color ) {
                Graphics gr = getOffScreenGraphics();
                gr.setColor( color );

                for ( int i=0; i < track.length-1; i++ )
                        gr.drawLine( track[i][0], track[i][1],
                                track[i+1][0], track[i+1][1] );

                repaint();
        }
```

```
        private void clearGr() {
                getOffScreenGraphics().clearRect(0, 0,
size().width, size().height);
        }

        public Graphics getOffScreenGraphics() {
                if ( drawGr == null ) {
                        drawImg = createImage( size().width,
size().height );
                        drawGr = drawImg.getGraphics();
                }
                return drawGr;
        }
```

classe tween.Canvas P. Niemeyer

Dans l'éditeur après appui sur le bouton tween, la méthode `setTracks()` est lancée puis "l'animation morphing" par `startTweening()`.
`setTracks()` positionne le champ `tracks` qui est le vecteur des formes à "joindre" par morphing.
`startTweening()` crée la thread de morphing dont le corps est `tweenTracks()`.

`tweenTracks()` va faire le morphing entre les 2 formes finales en parcourant le vecteur des formes finales (on peut supposer que ce vecteur n'a que 2 éléments !!).

Morphing (P. Niemeyer)

Finalement c'est donc la méthode `private void tweenTrack(int [][] fromTrack, int [][] toTrack)` qui est la clé de tout l'ensemble.
Cette méthode cherche d'abord la distance moyenne entre les 2 courbes. Cette distance donne alors le nombre de courbes intermédiaires à dessiner. Il faut alors construire et dessiner chaque courbe intermédiaire. Pour une courbe donnée (i.e. pour une valeur de `tweenNo`, on construit cette courbe à `len` points. len est calculé de sorte a être proche du nombre de point de la courbe finale pour les dernières courbes et vice-versa pour les premières courbes.
On repère les points correspondants au 2 courbes finales pour le point à construire, puis on construit ce point ce qui initialise les 2 composantes de `tweenTrack[i]`.

Les dessins sont fait en double buffering et on dessine 24 courbes par secondes.

Bibliographie

http://www.javaworld.com/javaworld/jw-03-1996/jw-03-animation.html

http://java.sun.com:81/applets/Fractal/1.0.2/example1.html

Teach yourself Java in 21 days : Laura Lemay, Charles L.Perkins ; ed Sams.net traduit en français ed S&SM "Le programmeur Java"